UNIVERSIDAD DE LA RIOJA

TRABAJO FIN DE ESTUDIOS

Título

Desarrollo de aplicaciones programables "System On Chip" de 32 bits, sobre FPGA.

Autor/es

HÉCTOR MANGADO SÁENZ

Director/es

JAVIER ESTEBAN VICUÑA MARTÍNEZ

Facultad

Escuela Técnica Superior de Ingeniería Industrial

Titulación

Grado en Ingeniería Electrónica Industrial y Automática

Departamento

INGENIERÍA ELÉCTRICA

Curso académico

2018-19

publicaciones.unirioja.es
E-mail: publicaciones@unirioja.es

UNIVERSIDAD DE LA RIOJA

ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA INDUSTRIAL

TRABAJO DE FIN DE GRADO

TITULACIÓN: Grado en Ingeniería Electrónica Industrial y Automática

CURSO: 2018/2019 **CONVOCATORIA: JULIO**

TÍTULO:

Desarrollo de aplicaciones programables "System On Chip" de 32 bits, sobre FPGA.

AUTOR: Héctor Mangado Sáenz

DIRECTOR/ES: Javier Esteban Vicuña Martínez

DEPARTAMENTO: Ingeniería Eléctrica

RESUMEN

El trabajo que se presenta a continuación trata del estudio y desarrollo de un Sistema On – Chip (SoC) dedicado, basado en un microcontrolador MicroBlaze de 32 bits, para el desarrollo de aplicaciones sobre dispositivos FPGA.

Se realiza dicho estudio asimilando en primer lugar los conocimientos de los componentes de un sistema de este tipo, diferenciando las partes del procesador MicroBlaze, la arquitectura mediante subsistemas y conexionado de BUS y los recursos propios de la FPGA, la cual dispone de una serie de módulos que se pueden añadir a un sistema personalizado o dedicado según se necesiten.

Una vez completado todo el estudio necesario y comprendida la teoría para el desarrollo de sistemas On Chip, se ha realizado de forma guiada, una aplicación demostrativa que utiliza varios de los distintos bloques o módulos periféricos que están disponibles en la tarjeta de desarrollo Nexys 4 DDR de Xilinx.

Por último , se desarrolla un Manual de Usuario completamente guiado para que cualquier usuario que lo desee, y tenga unos conocimientos mínimos de aplicaciones electrónicas y programación en lenguajes VHDL y C/C++, pueda realizar con éxito el proyecto que se proponga con unas características equivalentes a la demo que se desarrolla en este trabajo.

ABSTRACT

The project presented below deals with the study and development of a dedicated On-Chip System (SoC), based on a MicroBlaze 32-bit microcontroller, for the development of applications on FPGA devices.

This study is carried out by first assimilating the knowledge of the components of a system of this type, differentiating the parts of the MicroBlaze processor, the architecture through subsystems and BUS connection and the FPGA's own resources, which has a series of modules that can be added to a personalized or dedicated system as needed.

Once all the necessary study was completed and the theory for the development of On Chip systems was understood, a demonstrative application was carried out using several of the different blocks or peripheral modules that are available in the Nexys 4 DDR development card from Xilinx..

Finally, a fully guided User Manual is developed for any user who wants it, and has a minimum knowledge of electronic applications and programming in VHDL and C/C ++ languages, can successfully carry out the proposed project with equivalent characteristics to the demo that is developed in this project.

ÍNDICE

MEMORIA

Índice

Índice de Ilustraciones

Índice de tablas

Índice de Listados

0. Presentación

Este trabajo ha sido realizado por Héctor Mangado Sáenz, alumno de cuarto curso del Grado en Ingeniería Electrónica Industrial y Automática y se presenta al objeto de su reconocimiento académico para la obtención del título de Graduado por la Universidad de La Rioja.

1. Objeto principal

El objetivo principal del trabajo contempla tres aspectos principalmente:

El primero consiste en el análisis y estudio de la metodología de desarrollo de sistemas On – Chip programables sobre FPGA, basados en el microprocesador software MicroBlaze de 32 bits por medio de IP core MCS, sobre una tarjeta de desarrollo FPGA Nexys 4 DDR.

El segundo, correspondería con el desarrollo de una aplicación software que gestione los principales recursos disponibles en la tarjeta y que permita poner a prueba la comprensión del estudio efectuado y su aplicación en la programación y correcto funcionamiento de una aplicación demostrativa.

Por último, se pretende generar una documentación de todo el proceso por medio de la readacción de un manual de instrucciones de usuario que describa todas las fases del desarrollo de sistemas basados en MicroBlaze MCS.

2. Objetivos intermedios

Para la consecución del objetivo principal descrito en el apartado anterior, se han ido superando un conjunto de etapas intermedias, que se enumeran de forma breve a continuación:

- Estudio general de los sistemas On – Chip integrados
    - Análisis del flujo de desarrollo de los sitemas On – Chip integrados
    - Estudio del diseño hardware de plataformas FPRO para sistemas On – Chip
    - Adaptación de un sistema MicroBlaze para la tarjeta de Digilent, Nexys 4 DDR, estudio del sistema MCS (MicroBlaze)
    - Desarrollo un sistema portable para aprender el diseño de hardware y presentar la práctica de SoC
- Estudio del desarrollo de software de un sistema dedicado
    - Descripción general de la estructura y desarrollo del sistema
    - Estudio del mapeado de memoria y accesos a los registros de E/S
    - Desarrollo de controladores de dispositivos
    - Análisis del programa de pruebas y verificaciones
- Análisis del protocolo de bus FPRO y de las direcciones MMIO
    - Estudio del bus FPRO y su interfaz
    - Desarrollo de los distintos núcleos o bloques del sistema
    - Desarrollo del controlador MMIO
    - Estudio general del bus y puente MCS
- Desarrollo del bloque UART
    - Descripción general de la UART
    - Construcción y desarrollo del núcleo UART

    - Desarrollo del controlador UART
- Bloque XADC de Xilinx
    - Descripción general de XADC
    - Construcción y desarrollo del núcleo XADC
    - Desarrollo del controlador del dispositivo XADC
    - Sistema FPRO de prueba
- Estudio y desarrollo del bloque de Modulación por Anchura de Pulso (PWM)
    - Estudio del PWM como salida analógica
    - Análisis y diseño del PWM
    - Estudio del núcleo de desarrollo
    - Desarrollo del controlador PWM

- Estudio de los bloques antirrebote y multiplexación LED
    - Análisis y desarrollo del núcleo antirrebote
    - Análisis y desarrollo del núcleo de multiplexación LED

3. Alcance

Este trabajo contempla todas las fases necesarias para alcanzar los objetivos principales planteados en los apartados anteriores.

Como ya se ha citado en el apartado 1, uno de los objetivos principales es el estudio y análisis de un sistema On - Chip de 32 bits basado en MicroBlaze MCS para el cuál fue necesaria la búsqueda de información de cada una de las partes de su estructura completa. Una vez adquirida toda la información necesaria fue analizada y estudiada a conciencia para poder desarrollar un documento que facilite la comprensión de la creación de un sistema completo. La mayor parte ha sido obtenida del libro “FPGA PROTOTYPING BY VHDL EXAMPLES” de Pong P. Chu 2ª Edición, por lo que ha sido necesaria la lectura y comprensión completa de cada parte del sistema para poder realizar este documento y la ejecución de una demo que demuestra que se han alcanzado los objetivos del proyecto.

Durante el estudio de cada parte de la estructura, ha sido necesario realizar pruebas prácticas para la ayuda a la comprensión de cada núcleo o bloque que compone el sistema completo. Esta tarea práctica ha sido realizada con la tarjeta de Digilent Nexys 4 DDR siendo programada por medio del programa de Vivado perteneciente a Xilinx, con la cual se han realizado pruebas parciales por bloques para comprobar el completo entendimiento y funcionamiento de cada uno de los núcleos o “cores” para poder ser aplicado a un proyecto final tipo demo.

Las distintas opciones de aplicación utilizadas para la realización de este proyecto, dentro de las que ofrece la tarjeta mencionada, han sido el conjunto de 16 LEDs y 16 switches o interruptores, dos indicadores LED de color RGB, ocho displays de 7-segmentos, transmisión por medio de la UART, puerto de señal analógica Pmod para el uso del periférico XADC (convertidor de analógico a digital) y el conjunto de 5 botones de la propia tarjeta.

Por tanto, como indica el apartado 1 de objetivos principales, se ha realizado el estudio y análisis de un sistema On-Chip de 32 bits, teniendo que adquirir los conocimientos de plataformas FPRO para sistemas On-Chip, del sistema

MicroBlaze de 32 bits, de los protocolos de estructura y unión entre los distintos sistemas, del funcionamiento y aplicaciones de la tarjeta Nexys 4 DDR y del lenguaje y programación de los programas Vivado y SDK, para después poder realizar una aplicación tipo demo en la que se lleva a cabo todo lo aprendido de lo citado, y por último la realización de un Manual de Instrucciones de usuario para dar facilidad a la realización de aplicaciones de este tipo a personas con conocimientos básicos en programación de FPGA´s.

4. Antecedentes/Introducción

El trabajo que se presenta se centra en la realización de diseños de SoC (Sistemas On Chip) o Sistemas Integrados, basados en el microporcesador software Xilinx MicroBlaze de 32 bits, sobre la FPGA (matriz de puertas programable) de Xilinx DISPONIBLE EN LA TARJETA nEXYS 4 DDR.

La estructura de diseño de sistemas SoC se diferencia claramente en dos partes: diseño de la parte hardware (programación en lenguaje HDL) y parte software (programación en lenguaje C).

Los dispositivos HDL y FPGA permiten desarrollar y simular rápidamente un circuito digital sofisticado, realizarlo en un proceso de creación de prototipos y verificar el funcionamiento de la implementación física. A medida que la capacidad de los dispositivos FPGA continúa creciendo, un dispositivo puede acomodar un diseño SoC (Sistema On chip), que integra un procesador, módulos de memoria, periféricos de E / S y aceleradores de hardware personalizados en un solo chip.

Este documento utiliza un enfoque de para aprender a desarrollar este tipo de sistemas en base al intento de realizar pruebas y ejemplos e ilustra el proceso de desarrollo y diseño de FPGA y HDL en el contexto de SoC.

Los ejemplos conducen a un sistema integrado funcional con periféricos de E/S personalizados y aceleradores de hardware. Se presenta un marco de SoC simple, FPro (abreviado del título del libro "FPGA Prototyping"), como una plataforma para integrar todos los ejemplos de diseño juntos. Un sistema FPro contiene un procesador de núcleo suave Xilinx MicroBlaze MCS, un subsistema de video y el subsistema MMIO (mapeado de memoria de E/S) que puede incorporar bloques de E/S personalizados. A excepción del procesador, todos los componentes están diseñados y codificados desde cero.

El lenguaje HDL en sí no es el tema principal y su cobertura se limita a un pequeño subconjunto sintetizable. Se ilustran diseños más complicados y sofisticados en el contexto de SoC. A lo largo del presente documento se estudian muchos conceptos a nivel de sistema, incluida la derivación de un procesador de core soft y un sistema basado en IP (propiedad intelectual), la partición e integración de software y hardware, y el desarrollo de periféricos de E/S personalizados y aceleradores de hardware.

La práctica de codificación y diseño es "compatible hacia adelante", lo que significa que:

- La misma práctica puede aplicarse a grandes diseños en el futuro.

- La misma práctica puede ayudar a otras tareas de desarrollo del sistema, incluida la simulación, el análisis de tiempo, la verificación y las pruebas.

- La misma práctica se puede aplicar a la tecnología ASIC y diferentes tipos de dispositivos FPGA.

- El código puede ser aceptado por un software de síntesis de diferentes proveedores.

El público objetivo son los ingenieros en ejercicio que desean aprender sobre FPGA y el desarrollo basado en HDL. Los lectores deben tener un conocimiento básico de sistemas digitales, generalmente en programas de ingeniería de computación, y un conocimiento práctico del lenguaje C/C++.

5. Análisis de soluciones

5.1 DESCRIPCIÓN GENERAL DE SISTEMAS ON - CHIP Y DEL SoC PROPUESTO

En este apartado del documento se describen las técnicas de diseño de hardware para el desarrollo de un SoC integrado simple y funcional, que está formado principalmente por un subsistema de mapeado de memoria de entradas y salidas (MMIO) y un subsistema de video.

Centrando la atención en el diseño hardware, a continuación se comienza con la descripción general del concepto de SoC o Sistema Integrado.

5.1.1 SoC o SISTEMA INTEGRADO

5.1.1.1 Descripción general

Puede definirse como un sistema electrónico habitualmente basado en alguna arquitectura de microprocesador diseñado para realizar una o varias tareas específicas.

Dentro de los SoC, se pueden clasificar de acuerdo con varios criterios. Así, atendiendo al sistema informático que lo forman se pueden clasificar en dos grupos: sistema informático básico y sistema informático de uso general.

El primero no se trata del producto final, sino de una parte integrada de un sistema más grande al que le faltan una serie de componentes para llegar a ser el producto final deseado.

Un sistema informático de uso general, es el producto final el cual no necesita de más componentes adicionales para cumplir su función. Los programas de aplicación se desarrollan en base a los recursos disponibles del sistema informático de uso general.

Los sistemas SoC se utilizan en una amplia gama de aplicaciones y cada aplicación tiene sus propios requisitos específicos. Un sistema sencillo de control bajas prestaciones podría ser por ejemplo el que dispone un horno de microondas, donde solo implica una función de control simple y puede implementarse mediante un microcontrolador de 8 bits.

Por otro lado, un sistema de altas presataciones tiene una mayor complejidad, por ejemplo, el que requiere una cámara digital, la cual realiza varias tareas principales. De las más comunes podrían ser: por un lado, la gestión de las operaciones generales de E/S; por otro lado, la tarea de procesamiento de la

imagen y compresión de los datos para reducir el tamaño del archivo. La tarea requiere una cantidad significativa de tiempo de cálculo y un procesador integrado puede no ser suficientemente potente para ello, por tanto un circuito digital personalizado, como un acelerador de hardware, puede diseñarse para realizar esta tarea particular y quitar la carga de trabajo al procesador principal.

5.1.1.2 SoC programable (PSoC)

Se trata de la implementación de un sistema que contiene un procesador, y habitualmente periféricos de E/S simples y aceleradores de hardware específicos para manejar las operaciones de cómputo intensivo, en un dispositivo lógico programable, conocido como PLD (Programmable Logic Device).

La tecnología FPGA permite adaptar de forma versátil en lo que respecta a su capacidad, potencia de cálculo, dimensión y diversidad de todo el sistema. El procesador puede ser un bloque hardware disponible en el dispositivo FPGA, o ser programado por “software” consumiendo recursos programables del dispositivo. Esta segunda opción, permite seleccionar además de los periféricos de E/S, la interfaz de E/S personalizada y desarrollar aceleradores de hardware especializados para tareas de computación intensivas, adaptar el procesador a los requerimientos del SoC. Un sistema integrado FPGA proporciona una nueva dimensión de flexibilidad porque tanto el hardware como el software pueden personalizarse para satisfacer unas necesidades específicas. Se utiliza el término co-diseño hardware y software, para referirse al proceso de desarrollo en el que se diseña e implementa un SoC en el que se reparten las funcionalidades y prestaciones del sistema entre soluciones algorítmicas ejecutadas por el procesador y el diseño y desarrollo hardware de bloques y peiféricos específicos descritos mediante lenguajes de descripción de hardware (HDL), para las tareas más intensivas en hardware y de mayor coste computacional.

5.1.1.3 IP Core (Bloques de Propiedad Intelectual)

En el desarrollo de sistemas PSoC, cobra una gran relevancia la modularidad. Las herramientas disponibles y las metodologías que intervienen en su desarrollo permiten sacar una gran ventaja de la estandarización de los bloques que los conforman. Estos bloques periféricos son conocidos como IP cores: “bloques o macros de Propiedad Intelectual”. Nos referiremos como core, traducido como núcleo, aunque también puede ser nombrado como bloque o módulo. A lo largo del presente documento puede adoptar cualquiera de los nombres señalados que supondrá el mismo significado.

Estos bloques son funciones similares a las de una biblioteca de software, que están disponibles para ser utilizadas por las herramientas de desarrollo durante

el diseño de los SoC, y han sido desarrollados por los fabricantes de dispositivos, por terceros fabricantes o por los propios usuarios.

Los IP cores provistos por los fabricantes de las FPGA, generalmente se adaptan a sus propias plataformas propietarias y pueden requirir costes de licencia de uso. No son exportables a dispositivos de otros fabricantes y frecuentemente son considerados “cajas negras”, es decir, sin códigos fuente HDL.

Por lo tanto, el sistema debe ser rediseñado o modificado si se redirige a un dispositivo de un proveedor diferente.

5.1.2 FLUJO DE DESARROLLO DEL SoC

El diseño del SoC integrado consta de las siguientes fases:

- Partición de las tareas en rutinas de software y en bloques aceleradores de hardware (partición Hw/Sw).
- Diseño y desarrollo de los bloques o núcleos IP personalizados de usuario si fuera necesario.
- Desarrollo del hardware.
- Desarrollo del software.
- Implementación del hardware y del software, simulación y pruebas de depuración.

De las que se hablará en las siguientes subsecciones.

Debido a la complejidad de los sistemas digitales modernos, los IP core prediseñados se utilizan ampliamente en el desarrollo de SoC. Cada proveedor tiene su conjunto de IP cores, que proporciona una colección completa de IP core y controladores de dispositivos de software compatibles.

5.1.2.1 Partición hardware-software

El paso 1 (según el diagrama de la Ilustración 1) consiste en determinar la partición del hardware y del software.

En un diseño basado en SoC, una misma tarea o función puede implementarse por hardware o por software. Se trata de tomar decisiones a este respecto basadas en criterios como el rendimiento, la velocidad y la flexibilidad de las funiones implementadas.

Basándose en el requisito de rendimiento, la complejidad y la disponibilidad del núcleo del hardware, se puede decidir el tipo de implementación en consecuencia.

En la realidad, la mayoría de los diseños requieren de un cierto número de núcleos o bloques IP personalizados para aceleradores de hardware y periféricos de E/S especiales. El paso 2 (del diagrama de la Ilustración 1) es desarrollar los códigos de hardware y los controladores de software correspondientes de estos núcleos IP personalizados, desarrollados más adelante.

5.1.2.2 Flujo de desarrollo de hardware

Viendo el diagrama de la Ilustración 1, la rama izquierda representa el flujo de diseño de hardware. El paso 3 consiste en utilizar e integrar los núcleos IP para construir el sistema mediante la integración de IP (en Vivado Design Suite). Un usuario puede seleccionar núcleos IP, configurarlos con las características deseadas y conectar los bloques con una interfaz adecuada. El integrador IP invocará los recursos de la biblioteca y generará los códigos HDL. El integrador también produce un archivo de especificación de plataforma de hardware auxiliar, que contiene la “definición” del diseño SoC e incluye la configuración del procesador, tamaño y estructura de la memoria, los núcleos periféricos de E/S utilizados, la asignación de direcciones de memoria, etc.

El archivo HDL puede tratarse como un archivo normal y procesarse en consecuencia. Los pasos 4 y 5 del diagrama de la Ilustración 1 realizan la síntesis, la ubicación y el enrutamiento y, finalmente, generan el archivo de configuración FPGA, es decir, el archivo *.bit*.

5.1.2.3 Flujo de desarrollo de software

Representado en la rama derecha del diagrama de la Ilustración 1. Un programa de software de nivel superior normalmente contiene dos tipos de códigos. Un tipo son los “códigos del sistema”, que están diseñados de antemano y se proporcionan con el sistema. El otro tipo son los “códigos de aplicación”, que son desarrollados por el usuario para realizar las tareas personalizadas. Estas funciones del sistema y las rutinas del servicio son llamadas por los códigos de aplicación.

Un BSP (Board Package Support, o *paquete de soporte de placa*) es un proceso en el cual se encapsulan los códigos del sistema, dado que cada sistema tiene una estructura de memoria diferente y contiene un conjunto de periféricos de E/S. BSP es una colección personalizada de controladores de dispositivo y rutinas de inicialización que admite un sistema en particular. El origen del término placa (Board) proviene de los sistemas integrados anteriores que se implementaban en una placa de circuito impreso en lugar de en un solo dispositivo programable.

Un componente importante de los BSP son los controladores de dispositivo. Un controlador de dispositivo es un conjunto de rutinas que operan o controlan un dispositivo periférico en particular. Actúa como un “traductor” entre el hardware periférico y los programas de aplicación y permite que los programas de aplicación accedan a funciones periféricas sin necesidad de conocer detalles precisos.

El paso 6 es construir el BSP de acuerdo con la configuración del hardware SoC. La utilidad del constructor BSP examina la información central de IP, extrae los controladores de dispositivos construidos y las rutinas de inicialización de la biblioteca de software del proveedor, y crea el BSP para el diseño de SoC específico.

El paso 7 compila y vincula las rutinas de software y la biblioteca BSP y crea el archivo de imagen de software final, es decir, el archivo *.elf* (Executable and Linking File).

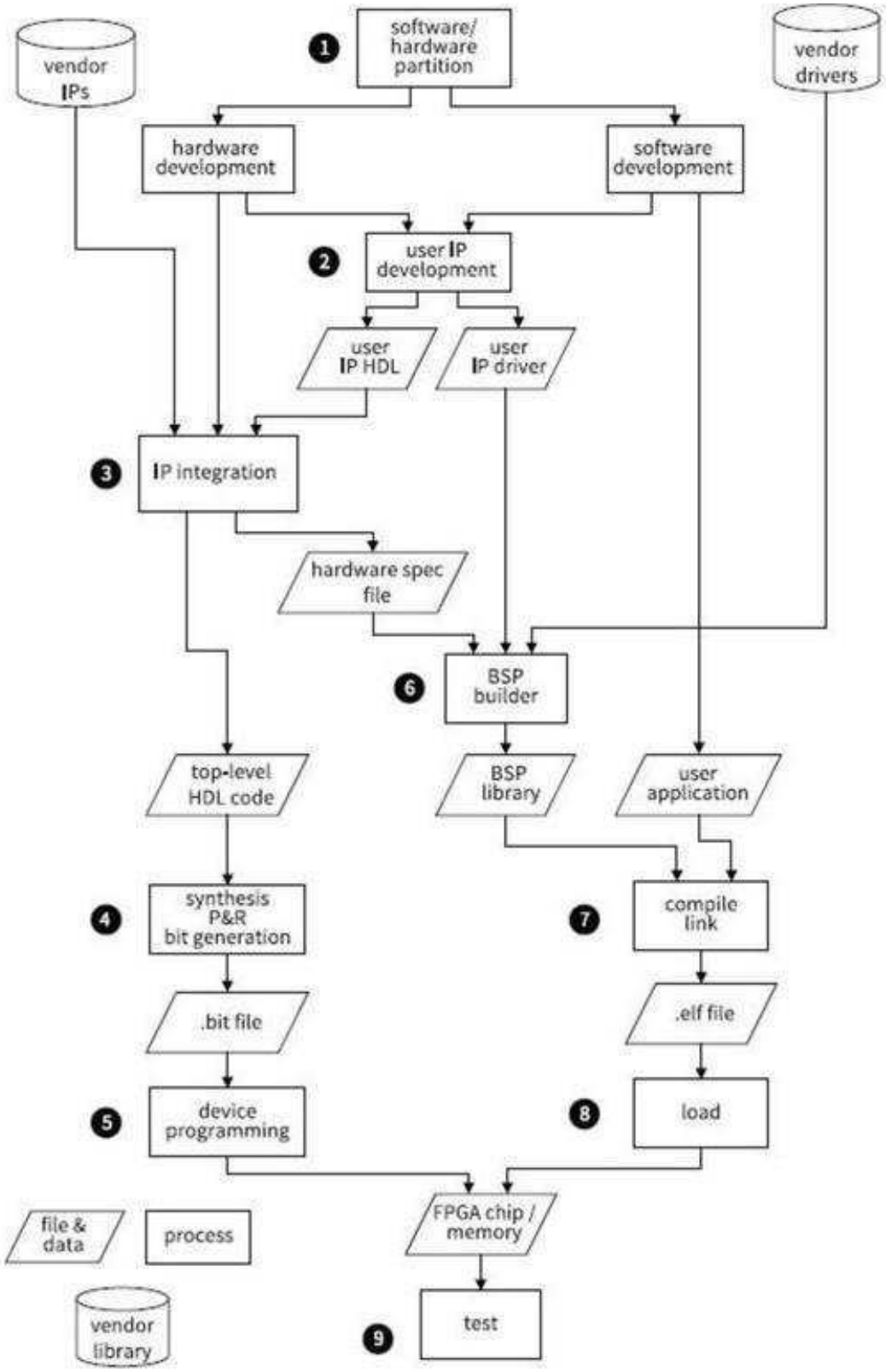


Ilustración 1: *Flujo de desarrollo de SoC centrado en IP.*

5.1.2.4 Implementación física y test

La implementación física del sistema implica dos pasos. En el primero se descarga el archivo de configuración FPGA al dispositivo FPGA y segundo se carga la imagen del software en la memoria principal del procesador. El sistema físico se puede probar después, como en el paso 9.

5.1.2.5 Desarrollo personalizado de núcleos IP

Por lo general, se tienen que diseñar núcleos IP personalizados para periféricos de E/S especiales o algoritmos de computación menos comunes. El desarrollo consta de tres tareas:

- Diseñar un circuito digital personalizado para implementar el algoritmo de cálculo o funcionalidad especial.
- Derivar una interfaz para conectar el circuito al bus o estructura de interconexión o marco IP del proveedor.
- Desarrollar un controlador de dispositivo para controlar el nuevo módulo de hardware e integrarlo en la biblioteca de software del proveedor

Las dos últimas actividades dependen de la plataforma IP del vendedor de FPGA. Se necesita estudiar cuidadosamente los protocolos de interfaz de la plataforma y la estructura del controlador para que el núcleo IP se pueda integrar en el marco del proveedor y se use en la utilidad de integración de IP. La interfaz y el controlador no son portátiles y deben ser rediseñados para cada proveedor.

5.1.3 PLATAFORMA FPRO SoC

5.1.3.1 Motivaciones

Como ya se describió, uno de los objetivos de este TFG consiste en diseñar una plataforma SoC simple, a la que llamaremos *FPro SoC (abreviatura de FPGA Prototyping), o simplemente FPro*. El sistema propuesto contiene un subsistema de mapeado de memoria de entradas y salidas (MMIO) y un subsistema de video. Dentro del contexto de SoC, las principales características de la plataforma FPro SoC son las siguientes:

- *Sencillo*. Define un protocolo de bus síncrono simple y una estructura de controlador de dispositivo sencilla. Se puede convertir a un núcleo IP agregando un circuito de interfaz simple y un controlador de dispositivo. El núcleo puede ser incorporado al sistema integrado existente.
- *Funcional*. La plataforma FPro SoC definida proporciona una variedad de periféricos de E/S e interfaces en serie de uso común (UART, SPI e I2C) e incluye controladores de dispositivos funcionales.

- *Portable*. Los núcleos IP de FPro SoC se desarrollan desde cero en HDL y no utilizan componentes privados de ningún proveedor. Los núcleos IP y los códigos de software son portables y pueden reutilizarse para diferentes dispositivos FPGA y placas de prototipos.

- *Compatible de forma ascendente*. El desarrollo sigue principios y prácticas de diseño rigurosos y probados. Los núcleos IP y los controladores desarrollados pueden modificarse fácilmente para incorporarlos en los marcos comerciales de IP existentes.

- *Divertido*. Puede incorporar módulos de E/S existentes y desarrollar rápidamente un proyecto de creación de prototipos funcional. Además, esta plataforma puede proporcionar capacidad de aceleración de hardware y, por lo tanto, es más capaz y más flexible que cualquier placa de microprocesador. Esto da la oportunidad de desarrollar proyectos interesantes y desafiantes y hacer que el estudio de hardware sea más “divertido”.

5.1.3.2 Organización del hardware de la plataforma

El diagrama de nivel superior de un sistema FPro mostrado en la Ilustración 2, se compone de cuatro partes principales:

I.- Módulo procesador

II.- Puente FPro y bus FPro

III.- Subsistema MMIO (E/S mapeado en memoria)

IV.- Subsistema de video

Solo se utilizarán los núcleos IP del proveedor Xilinx para el bloque procesador, para el controlador de memoria, el búfer de línea y el circuito de administración del reloj. Todos los demás núcleos se construyen desde cero.

I.- Módulo procesador*:* consta de un procesador, un núcleo de controlador de memoria y RAM. Se construye a partir de los núcleos IP del proveedor y para ser utilizado en la plataforma de SoC de FPro, el núcleo debe presentar las siguientes características:

- Bus de datos de 32 bits de ancho.

- Espacio de direcciones de memoria de 32 bits.

- Esquema de E/S mapeado en memoria para acceso de E/S.

Casi todos los procesadores basados en FPGA soportan estas características. Pueden ser módulos de memoria interna de FPGA o dispositivos de memoria externos.

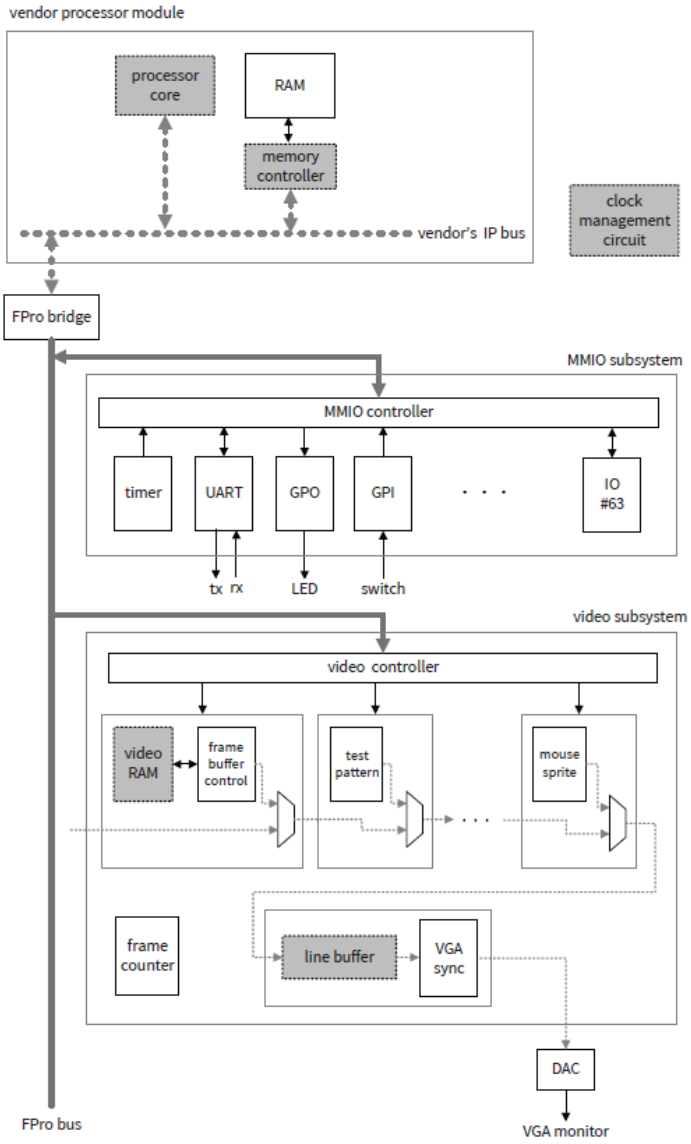


Ilustración 2: *Diagrama de alto nivel de un sistema FPro.*

II.- Puente FPro y bus FPro: el procesador necesita comunicarse con otros núcleos. Esto se realiza mediante una estructura de bus o interconexión especificada en la plataforma IP del proveedor. En este caso se define un protocolo simple de bus síncrono para los dos subsistemas y lo llamamos bus FPro. El puente FPro convierte las señales de bus nativas del proveedor en señales de bus FPro.

III.- Subsistema MMIO: la memoria y los registros de los periféricos de E/S se asignan al mismo espacio de direcciones. Significa que el procesador no hace

distinciones entre la memoria y los periféricos de E/S y utiliza las mismas instrucciones de lectura y escritura para acceder a los periféricos de E/S.

El subsistema MMIO proporciona un espacio direccionable para alojar periféricos de E/S especiales y de propósito general asignados en memoria, así como aceleradores de hardware y consta de un controlador para seleccionar una ranura (dirección base) específica y puede acomodar hasta 64 módulos instanciados. Después de ser “envuelto” con un circuito de interfaz, la lógica digital personalizada se puede insertar en la plataforma FPro.

IV.- Subsistema de video: establece un marco para coordinar la operación de los núcleos de video. Un núcleo de video genera o procesa el flujo de datos de video. Los núcleos están dispuestos como una cadena en cascada. El flujo de datos se canaliza y se “mezcla” en cada etapa y, finalmente, se muestra en un monitor VGA. Los datos se generan continuamente y pasan a través de una cadena de componentes para su procesamiento***.***

5.1.3.3 Organización del software de la plataforma

Se utiliza un esquema de software simple para el sistema, lo que significa que no utilizará sistema operativo. En su forma más simple, el procesador se inicia directamente en un bucle principal infinito, que contiene funciones para verificar las entradas, realizar los cálculos y escribir salidas.

Como muestra su jerarquía en la Ilustración 3, contiene una ***capa de hardware***, una ***capa de controlador*** y una ***capa de aplicación***. Una *rutina de arranque* asociada con el procesador, realiza en primer lugar el proceso de inicialización, en el que se borran los cachés, se configuran los segmentos de pila, y se inicializan las interrupciónes, y luego transfiere el control al programa principal. Los códigos para realizar esta rutina de arranque son obtenidos del diseñador del procesador y todos los demás controladores de dispositivos son construidos desde cero.

Ilustración 3: *Jerarquía de software de un sistema FPro SoC.*

Se desarrollan varias rutinas de utilidad sencillas, para facilitar el desarrollo del software, que mantienen la hora del sistema y ayudan a mostrar un mensaje de depuración en la consola. Se utilizan para ello los bloques del temporizador y la

UART, en las ranuras 0 y 1. Siempre deben ser instanciados en las dos primeras ranuras y no ser remplazados.

Cada núcleo de E/S en el sistema FPro está acompañado por un controlador. Se selecciona C++ para el desarrollo de los controladores debido a su compatibilidad con la encapsulación de datos. Se creará una clase de C++ para cada núcleo.

Excepto para acceder a la hora del sistema y enviar mensajes de depuración, una clase es en gran parte “independiente” y no interactúa con otras clases. Cuando se adjunta o se elimina un módulo de un sistema FPro, los archivos del controlador correspondiente deben incluirse o eliminarse de los proyectos de software. En el programa de aplicación principal, se creará una instancia para cada bloque de IP instanciado y los métodos en la clase se utilizarán para acceder y controlar el bloque. El “estado” del módulo, si existe, se mantiene dentro de la sección privada del objeto instanciado y no implica variables externas.

5.1.3.4 Flujo de desarrollo modificado

El flujo de desarrollo original de la Ilustración 1 debe revisarse ya que se ve modificado para adaptarse a la plataforma SoC de FPro, como se muestra en la Ilustración 4. Si bien el procedimiento básico permanece sin cambios, se necesita construir manualmente el código HDL de nivel superior e incluir manualmente los archivos del controlador del dispositivo en la aplicación de software.

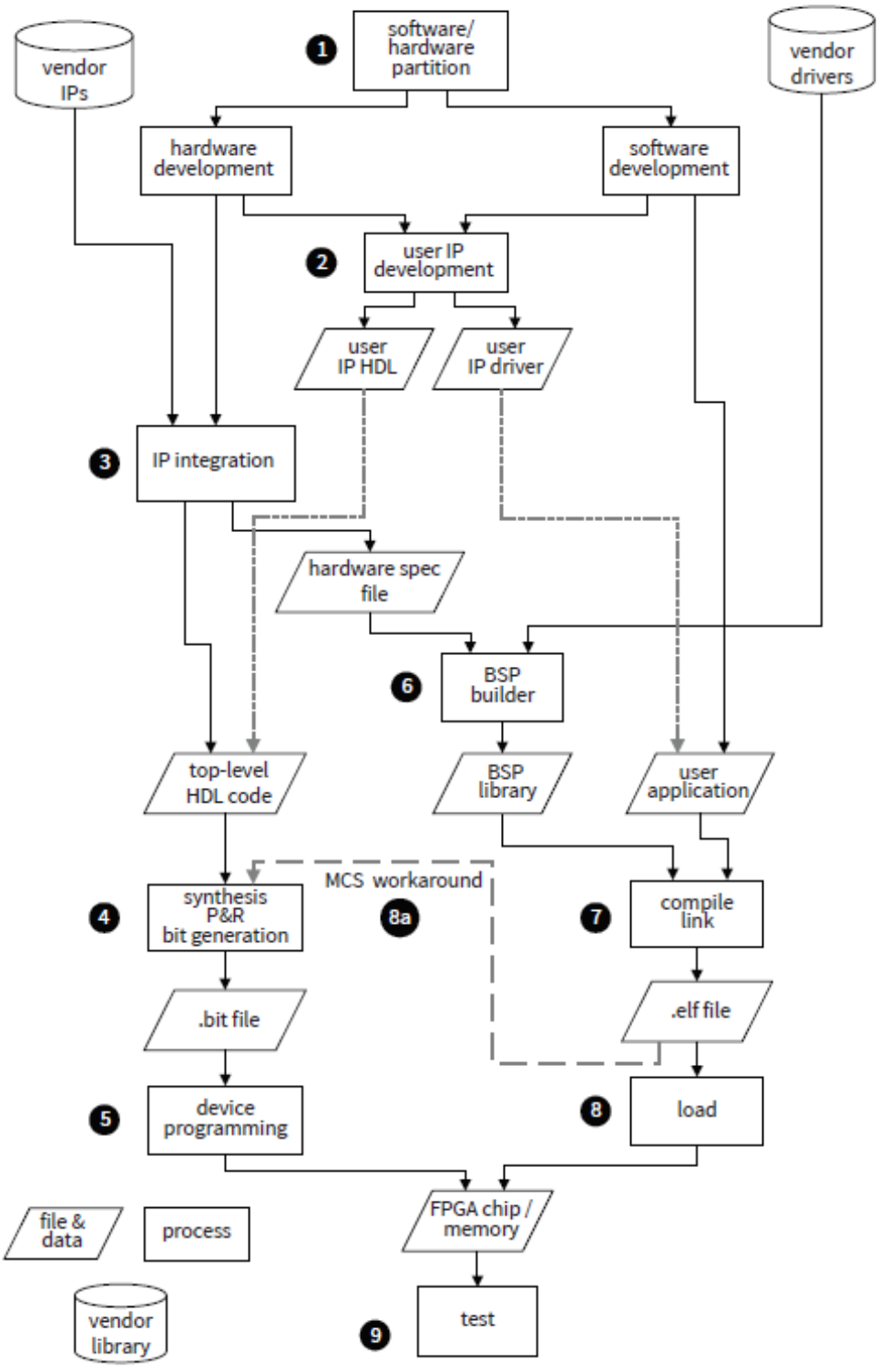


Ilustración 4: *Flujo de desarrollo de SoC de FPro.*

Los principales cambios son los siguientes:

- En el paso 3, sólo el módulo del procesador se genera a través de la utilidad de integración de IP. Se debe construir manualmente el código HDL para el sistema de nivel superior, que se compone de la creación de instancias del módulo del procesador generado anteriormente y los subsistemas de video y MMIO del paso 2.

- En el paso 6, solo los códigos relacionados con el procesador, como la rutina de arranque, se incluirán en la biblioteca BSP. Se deben examinar manualmente los IP cores en el archivo HDL de nivel superior e incluir los archivos de controlador correspondientes en el proyecto de software de la aplicación.

- Dado que el módulo del procesador la mayor parte del tiempo, los pasos 1 y 6 solo necesitan ser ejecutados una sola vez. Los archivos HDL generados y la biblioteca BSP pueden ser usados en diseños posteriores.

5.1.4 ADAPTACIÓN EN LA TARJETA DIGILENT NEXYS 4 DDR

En este proyecto se utiliza la placa de prototipos Digilent Nexys 4 DDR, diseñada alrededor del dispositivo Xilinx Artix 7 XC7A100T. Este dispositivo posibilita los recursos necesarios para programar un procesador software, conocido como MicroBlaze, así como un sistema completamente “pre configurado”, conocido como MicroBlaze MCS (para MicroBlaze Micro Controller System).

MicroBlaze es un procesador de 32 bits con arquitectura RISC (computadora con conjunto de instrucciones reducido). Es altamente configurable y puede incorporar una unidad de coma flotante opcional, cachés de instrucciones y datos, unidad de administración de memoria, etc.

Utiliza principalmente los protocolos AXI de ARM para interactuar con otros núcleos IP. Centenares de núcleos IP de Xilinx y de terceros proveedores pueden integrarse para formar parte de un diseño de SoC. El flujo de la Ilustración 1 está dirigido a este tipo de configuración.

MicroBlaze MCS es un sistema informático completo que consta de un procesador MicroBlaze pre configurado, una memoria RAM construida con bloques de memoria interna de la FPGA y un módulo de E/S con un conjunto estándar de periféricos. Proporciona un grado limitado de configuración. Un usuario puede configurar el tamaño de RAM (entre 8 KB y 128 KB) y seleccionar un pequeño conjunto de periféricos de E/S simples.

Como aspecto negativo, el paso 8 de la Ilustración 4 no funciona correctamente. La solución es asociar el archivo *.elf* como los “valores iniciales” de la memoria

interna de FPGA y regenerar el archivo de configuración (archivo *.bit*), que se muestra con una línea gruesa discontinua en la parte inferior de la misma figura. No parece previsible que Xilinx resuelva este inconveniente en futuras versiones de Vivado.

El flujo de revisión se convierte en los siguientes pasos:

- Desarrollar e implementar el hardware (Pasos 1 a 4).
- Desarrollar e implementar el software (Pasos 2 a 7).
- Asociar el archivo *.elf* en el proyecto de hardware (Paso 8).
- Regenerar el archivo de configuración *.bit* con el archivo *.elf* incorporado (repitiendo el paso 4).
- Programar el dispositivo FPGA y realizar pruebas (pasos 5 y 9).

5.1.5 PORTABILIDAD del sistema

Uno de los objetivos principales de este TFG consiste en revisar el proceso completo que permite desarrollar de forma práctica el hardware y software asociado a un SoC. El desarrollo de un sistema SoC basado en FPGA conlleva cierta dependencia con el dispositivo FPGA y con la tarjeta de desarrollo, lo que dificulta su portabilidad. En los siguientes apartados se discute la portabilidad de los diferentes bloques, para ser desarrollados en otras familias de FPGAs, o plataformas de desarrollo de otros fabricantes:

5.1.5.1 Módulo procesador y puente

El módulo procesador en este caso Microblaze MCS, está construido a partir de los bloques IP privados del proveedor Xilinx, y potencialmente introduce varios inconvenientes de portabilidad en diferentes aspectos:

- Procesador.
- Controlador de memoria y RAM.
- Interfaz y puente.
- Software de carga y arranque.

La plataforma FPro requiere un bloque de procesador de 32 bits que admita el direccionamiento de E/S mapeado en memoria. Por este motivo, las configuraciones de memoria que utiliza el IP core MCS, no causarán problemas de compatibilidad insalvables. La forma más sencilla de crear los módulos de

procesador es utilizar la memoria interna de FPGA, tal como lo implementa MicroBlaze MCS.

El protocolo de BUS FPro está diseñado para transacciones simples de lectura y escritura síncrona, sin ráfagas. Esto facilita el diseño de un bloque dedicado a hacer las funciones de puente.

Al compilar y vincular el código de software, no existe un procedimiento estándar para cargar un archivo *.elf* (Paso 8 en el flujo de desarrollo). Esto depende del dispositivo, la configuración de la memoria, la placa de prototipos y la plataforma de desarrollo de software. Se necesita consultar el manual específico para completar esta tarea.

5.1.5.2 Subsistema MMIO

Los códigos HDL son completamente portátiles. Se pueden implementar siempre que la FPGA y la tarjeta de desarrollo tengan los periféricos externos adecuados. La única excepción es el ADC incorporado (convertidor de analógico a digital) de los dispositivos de la familia 7 de Xilinx,Artix, Kintex, Zynq, etc. conocido como XADC, que solo está disponible para este tipo de programables de Xilinx.

Dado que la tarjeta Nexys 4 DDR contiene todos los periféricos necesarios, para el sistema FPro, todos los núcleos IP de MMIO se pueden implementar y probar sin añadir ningún componente externo adicional. Los esquemas para estos periféricos se pueden encontrar en el manual en línea de la tarjeta Nexys 4 DDR indicado en el apartado correspondiente a *“ESQUEMAS”* y se pueden reconstruir en consecuencia.

5.1.5.3 Subsistema de video

La mayor parte del subsistema de video está diseñado a bajo nivel, excepto tres componentes: el circuito de gestión de reloj, el búfer de línea y el búfer de trama, que utilizan núcleos IP del proveedor. El circuito de gestión del reloj y el búfer de línea admiten la sincronización VGA, cuya frecuencia de reloj es diferente de la frecuencia de reloj del sistema. El primero requiere una macro de tipo PLL (bucle de enganche de fase) y la última se basa en una macro de búfer FIFO de doble reloj. Los macros se pueden instanciar mediante código HDL de forma directa. Por lo tanto, el circuito de gestión de reloj y el búfer de línea no plantearían tampoco serios problemas de portabilidad.

El búfer de trama tiende a ser el núcleo IP más problemático y menos portátil en la estructura de FPro. La parte clave del búfer de trama es una memoria de doble puerto a la que acceden el procesador y el control de trama. El búfer requiere una cantidad sustancial de RAM y, por lo tanto, debe ser implementado por dispositivos de memoria externos a la FPGA. Esto plantea varios problemas:

- Las placas de prototipos FPGA tienen diferentes tipos de dispositivos de memoria y configuraciones y algunas placas más simples pueden no tener ninguno.
- A excepción de los dispositivos SRAM simples, se necesita un núcleo IP de controlador de estos tipos de memoria de cierta complejidad.
- El control del búfer de trama deben interactuar con el controlador de memoria propietario e implementar el circuito de control de acceso de doble puerto.
- El mismo dispositivo de memoria externa se puede usar como memoria RAM del procesador y búfer de trama al mismo tiempo. La partición complica aún más la interfaz y la configuración.

Por lo tanto, es difícil construir un búfer de trama susceptible de ser usado en otras plataformas diferentes. Es el bloque más complicado en términos de su portabilidad.

5.1.5.4 Organización

En esta parte se proporciona una descripción general de la arquitectura hardware del sistema y del desarrollo de software integrado a través de la construcción de un sistema llamado vanilla FPro, que se refiere a un sistema general FPro pero minimizado a un subsistema MMIO y 4 núcleos o bloques de E/S para facilitar el estudio y los ejemplos, y que contiene un bloque temporizador, un bloque UART, un bloque GPI (entrada de propósito general) y un bloque GPO (salida de propósito general). El diagrama conceptual de bloques se muestra en la Ilustración 5.

La siguiente parte muestra cómo diseñar una matriz de núcleos MMIO para los periféricos en la tarjeta Nexys 4 DDR, que incluye un bloque PWM (modulación por anchura de pulso), un bloque antirrebote, un bloque de displays de siete segmentos y un controlador Xilinx XADC.

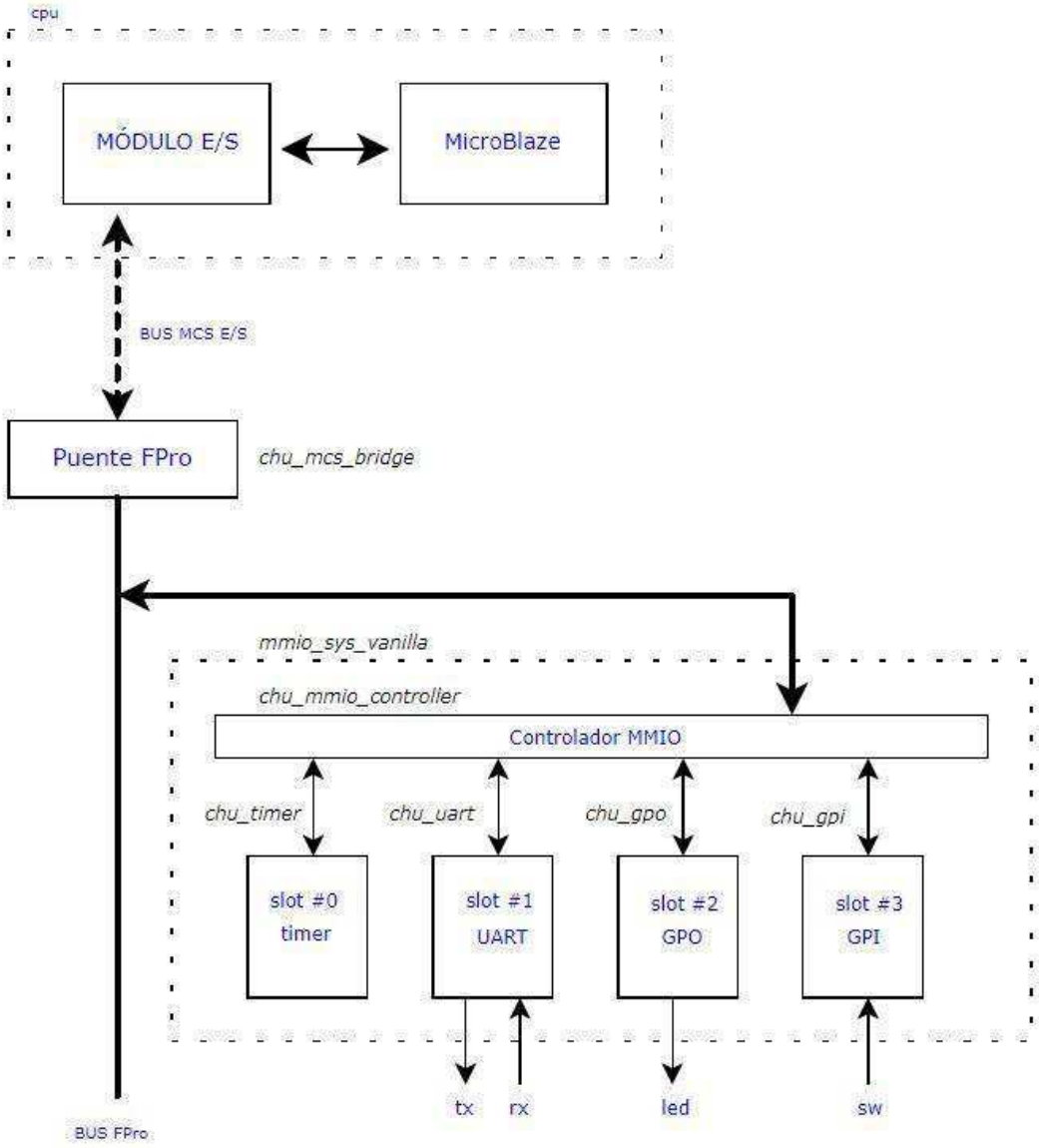


Ilustración 5: *Sistema Vanilla FPro.*

5.2 DESARROLLO DEL SOFTWARE DEL SISTEMA DEDICADO

Se va a hacer una descripción general del desarrollo de software para un sistema embebido dedicado, desarrollado a bajo nivel y se va a demostrar cómo estructurar un código sólido y ordenado para controlar y acceder a diversos periféricos de E/S.

5.2.1 DESCRIPCIÓN GENERAL DEL DESARROLLO DEL SISTEMA DEDICADO

5.2.1.1 Sistema similar a una computadora de escritorio VS sistema dedicado

El desarrollo de software para un sistema microprogramable dedicado y un sistema completo que utiliza un sistema operativo, es muy diferente. Un sistema operativo completo se ejecuta de forma continua y sirve una capa intermedia que enmascara los detalles del hardware al programa de aplicación. La jerarquía simplificada se muestra en la Ilustración 6 (a).

En un sistema completo, antes de iniciar la función `main()` de un programa, el sistema operativo (host) prepara el entorno de tiempo de ejecución asignando los recursos necesarios e inicializando los servicios del sistema. Esto significa que el programa de aplicación puede asumir que todas las bibliotecas y los servicios de E/S están listos para usar y que no se necesita ningún trabajo adicional.

Por el contario, un sistema dedicado, constituye un entorno independiente. Solo se admite un conjunto mínimo de bibliotecas C predefinidas, principalmente relacionadas con el tipo de datos y las definiciones de constantes. Si se requiere un servicio, el programa de aplicación debe configurar la manipulación del dispositivo por sí mismo.

En el “escenario más simple”, como se muestra en la Ilustración 6 (c), el programa de aplicación debe crear los servicios de E/S necesarios desde cero manipulando directamente los registros de los dispositivos de ES. En enfoque más adecuado es desarrollar un controlador simple para cada dispositivo de E/S y acceder a cada dispositivo a través de su controlador, como se muestra en la Ilustración 6 (b).

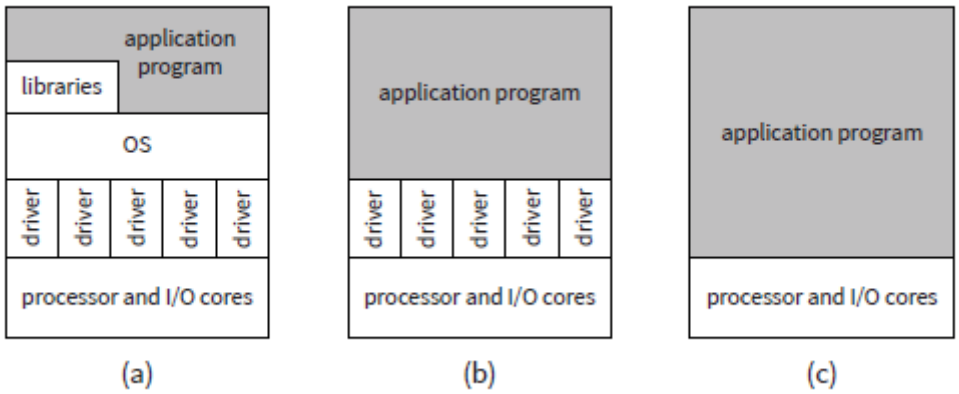


Ilustración 6: *Jerarquía software.*

5.2.1.2 Arquitectura básica del programa embebido

Una aplicación incorporada consiste en una colección de tareas, implementadas por aceleradores de hardware, rutinas de software o ambos.

A diferencia de una aplicación de escritorio, un programa incrustado puede ejecutarse continuamente y no termina.

El sistema ejecuta la función `sys_init()` una vez para realizar la inicialización y luego ingresa al súper bucle e invoca las funciones de la tarea por turnos. Funciona correctamente si el tiempo total de ejecución del bucle es pequeño y si cada tarea puede invocarse de manera oportuna.

5.2.2 E/S MAPEADO EN MEMORIA

5.2.2.1 Descripción general

Un periférico de E/S generalmente contiene un conjunto de registros para los comandos, el estado y los datos. Un procesador utiliza el mismo espacio de direcciones para acceder a la memoria y también para acceder a dispositivos de E/S. Por tanto, las instrucciones de carga y almacenamiento utilizadas para acceder a la memoria también se pueden usar para acceder a dispositivos de E/S. Cuando se construye un SoC, se asignan porciones de espacio de memoria al módulo RAM y los núcleos de E/S. El espacio de direcciones de memoria de un sistema informático simple que contiene un núcleo de temporizador y un núcleo de UART se muestra en la Ilustración 7.

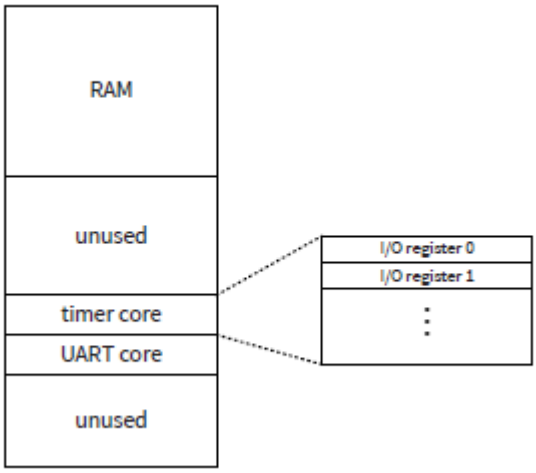


Ilustración 7: *Mapa de direcciones de un sistema simple.*

La dirección de un “slot” o ranura asignado se conoce como la dirección base. A la dirección de una palabra de memoria específica o de un registro E/S, se puede acceder agregando un “OFFSET” a la dirección base.

Nos referiremos como slot, traducido como ranura, aunque también puede ser nombrado como espacio de memoria. A lo largo del presente documento puede adoptar cualquiera de los nombres señalados que supondrá el mismo significado.

5.2.2.2 Alineación de la memoria

Un procesador de 32 bits significa que los datos se transfieren y procesan en una unidad de 32 bits. Se define una palabra como una unidad de 32 bits (cuatro bytes). Un procesador de 32 bits normalmente tiene un bus de direcciones de 32 bits y, por tanto, puede acceder a 2^{32} posiciones de memoria.

El espacio de direcciones generalmente se representa en términos de bytes (“byte direccionable”) y el bus de direcciones de 32 bits implica un espacio direccionable de 2^{32} bytes. Para acomodar datos de 32 bits, se agrupan cuatro bytes de memoria para formar una palabra. Los cuatro bytes se alinean en una dirección de memoria que es múltiplo de cuatro e implica que los dos LSB de la dirección del byte inicial en una palabra siempre son 00 y se puede acceder a una palabra utilizando los 30 MSB de la dirección. La memoria se puede tratar como un espacio direccionable de 32 bits o como un espacio direccionable de 30 bits.

5.2.2.3 Mapa de registro de E/S

Un bloque o unidad de E/S se presenta al sistema como una colección de registros que constituyen el propio “espacio de memoria direccionable” del bloque. El valor de la dirección se conoce como el desplazamiento u OFFSET. El formato de datos y la funcionalidad de cada registro son diferentes. Un mapa de registro de E/S representa las propiedades y los campos de estos registros.

En la Ilustración 8 se muestra un mapa de registro de E/S del bloque del temporizador. Hay tres registros. El número izquierdo muestra el valor del OFFSET o desplazamiento (que es 0, 1 y 2) y los caracteres correctos indican el tipo de acceso posible, que puede ser `r`, `w` o `r/w` (para lectura, escritura o lectura y escritura).

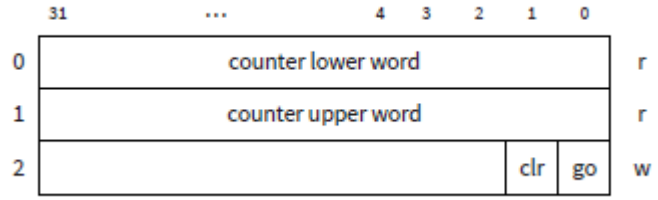


Ilustración 8: *Mapa de registro de E/S de un bloque temporizador.*

Después de asignar una dirección base a un bloque de E/S, sus registros se convierten en parte del espacio de memoria. El procesador puede acceder a un registro específico sumando el desplazamiento a la dirección base.

5.2.2.4 Espacio de direcciones de E/S del sistema FPro

El sistema FPro consta de dos subsistemas (MMIO y Video) y diversos bloques IP. Para lograr la portabilidad, se combinan en un único espacio de direcciones y los conectamos a la estructura del procesador a través de un bloque “puente”. Es decir, los dos subsistemas, junto con todos sus núcleos, se tratan como un único módulo de E/S. El puente y el controlador interno del sistema FPro realizan la decodificación y multiplexación para acceder a un núcleo individual y sus registros de E/S.

Los subsistemas combinados requieren un espacio de direcciones de bytes de 24 bits y aparecen como un módulo de E/S con 2^{22} registros de 32 bits. El espacio solo cuenta con $\frac{1}{2^8}\left(\frac{2^{24}}{2^{32}}\right)$ del espacio total de direcciones del procesador, lo que no debería suponer un problema. Se usa el MSB (el bit 23 de la dirección del byte) para diferenciar los dos subsistemas, con 0 para el subsistema MMIO y 1 para el subsistema de Video.

La asignación de direcciones para el subsistema MMIO se define de la siguiente manera:

- El subsistema proporciona 64 slots para conectar hasta 64 (2^6) bloques de E/S.
- Cada bloque de E/S se asigna con 32 (2^5) registros.
- Cada registro tiene 32 bits de ancho (una palabra).

Sin embargo, la mayoría de los núcleos de E/S no utilizarán los 32 registros y no siempre necesitarán 32 bits de datos.

5.2.3 ACCESO DIRECTO AL REGISTRO E/S

El acceso a un registro de E/S consiste en leer o escribir directamente una palabra en una dirección de memoria específica. Esto se puede hacer a través del tipo de datos de puntero de C.

5.2.3.1 Revisión del puntero C

Un puntero C almacena una referencia a un objeto. Una variable de puntero se designa con `*`, como por ejemplo en `int *ptr`. Esto indica que `ptr` es un puntero y hace referencia a una ubicación en la que se almacena un entero (tipo de datos `int`). Por lo tanto, una referencia y un puntero son solo una forma implícita y abstracta de representar una dirección de memoria.

El funcionamiento del segmento de código se ilustra en la Ilustración 9.

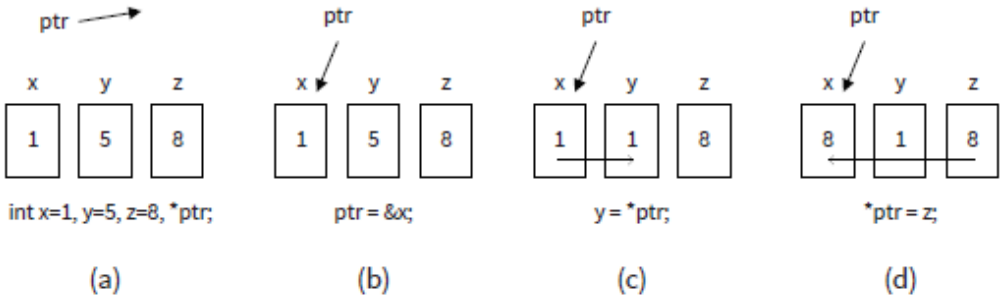


Ilustración 9: *Instantáneas de operaciones de puntero.*

Se usa una flecha para indicar que `ptr` es una variable de puntero. Dos operadores unarios, & y `*`, están aSoCiados con operaciones de puntero. El operador & devuelve la dirección de una variable y se conoce como el operador de dirección. El operador `*` devuelve el contenido señalado por el puntero y se conoce como el operador de desreferencia.

El valor de una variable de puntero generalmente se manipula implícitamente, como lo ilustra el segmento anterior. El valor real de `ptr` es dependiente del sistema. Generalmente no se debe conocer el valor explícito.

5.2.3.2 Puntero C para registro de E/S

En un sistema dedicado, diseñado a bajo nivel, a cada registro de E/S se le asigna una dirección de memoria, que se puede considerar como un valor de un puntero. A diferencia del puntero analizado antes, se conoce el valor explícito de la dirección y se debe utilizar este valor para acceder al registro. Debido a la complejidad y la dificultad de comprensión del código, se visualiza con modificaciones en la sección 5.2.4.3.

5.2.4 ACCESO DE REGISTRO DE E/S ROBUSTO

Se introduce a continuación un método robusto y ordenado para pasar información de asignación de dirección entre el hardware y el controlador de software y para acceder a los registros de E/S.

Se trata de tres archivos de cabecera C y un archivo de declaración de paquete VHDL:

- `chu_io_map.h`: contiene la información correspondiente al mapeo de direcciones para el código C/C++.
- `chu_io_map.vhd`: contiene la información de mapeo de direcciones para el código VHDL.
- `inttypes.h`: proporciona tipos de datos de bajo nivel definidos explícitamente.
- `chu_io_rw.h`: proporciona macros de lectura y escritura a registro de E/S.

5.2.4.1 `chu_io_map.h` y `chu_io_map.vhd`

La asignación es fija para los slots o direcciones base en el subsistema MMIO y los módulos en el subsistema de video. Sólo dos partes se pueden cambiar:

- La dirección base del puente
- La asignación de slots en el subsistema MMIO

Se expresa la información utilizando constantes simbólicas y se guardan en `chu_io_map.h` y `chu_io_map.vhd`. El primero es un archivo de encabezado C/C++ utilizado para el desarrollo software y el último contiene una declaración de paquete VHDL para usarse junto con el desarrollo de hardware.

El procesador trata el MMIO y los subsistemas de video como un único módulo de E/S y se comunica con esos módulos a través del puente. La dirección base del puente es la dirección de inicio del módulo y se asigna cuando se crea el sistema. Para la configuración de MicroBlaze MCS, se trata de un valor fijo de valor `0xC0000000`.

Para un núcleo de IP conectado al subsistema MMIO, su dirección base se puede calcular utilizando el número de slot asignado. Hay que recordar que cada ranura contiene 32 palabras (128 bytes). La dirección base de la ranura n es:

```
Bridge_base_address + n*32*4
```

La asignación de slots indica qué tipo de núcleo de IP se adjunta a una ranura en particular. Se utilizan un conjunto de constantes simbólicas que transportan la configuración de MMIO.

Además de la información de asignación de direcciones, `chu_io_map.h` incluye una constante simbólica para la frecuencia de reloj del sistema. El segmento de código para el sistema FPro se muestra en el listado 2.1.

Listado 2.1. Definición de constantes y ranuras `(chu_io_map.h)`

```
#ifndef _CHU_IO_MAP_INCLUDED
#define _CHU_IO_MAP_INCLUDED

#ifdef __cplusplus
extern "C" {
#endif

//////////////////////////////////////////////////////////////////*/
// #ifdef _NEXYS4

// system clock rate in MHz; used for timer and uart
#define SYS_CLK_FREQ 100

//io base address for microBlaze MCS
#define BRIDGE_BASE 0xc0000000

// slot module definition
// format: Slot#_ModuleType_Name
#define S0_SYS_TIMER  0
#define S1_UART1      1
#define S2_LED        2
#define S3_SW         3
#define S4_USER       4
#define S5_XDAC       5
#define S6_PWM        6
#define S7_BTN        7
#define S8_SSEG       8
#define S9_SPI        9
#define S10_I2C      10
#define S11_PS2      11
#define S12_DDFS     12
#define S13_ADSR     13
```

```
// video module definition
#define V0_SYNC      0
#define V1_MOUSE     1
#define V2_OSD       2
#define V3_GHOST     3
#define V4_USER4     4
#define V5_USER5     5
#define V6_GRAY      6
#define V7_BAR       7

// video frame buffer
#define FRAME_OFFSET 0x00c00000
#define FRAME_BASE  BRIDGE_BASE+FRAME_OFFSET

// #endif    // _NEXYS4
/*************************************************************************/

#ifdef __cplusplus
} // extern "C"
#endif

#endif  // _CHU_IO_MAP_INCLUDED
```

Un paquete VHDL contiene definiciones de constantes, definiciones de tipos de datos y subprogramas para compartir en múltiples diseños. Se crea un paquete personalizado que se define en el archivo `chu_io_map.vhd` y sus detalles se tratarán en la sección 5.3.5.1. Una parte del paquete contiene la información de definición de slots y el segmento de código correspondiente es:

```
constant    S0_SYS_TIMER  :   integer   :=  0;
constant    S1_UART1      :   integer   :=  1;
constant    S2_LED        :   integer   :=  2;
constant    S3_SW         :   integer   :=  3;
```

Un número de slot desajustado entre el hardware y el software provoca errores graves de compilación y funcionamiento del programa.

5.2.4.2 Inttypes.h

El lenguaje C tiene muchos tipos de datos predefinidos. El ancho (número de bits) de cada tipo de datos se deja al compilador y a la implementación. A menudo es importante conocer el ancho exacto y el formato de los registros y los datos. Para facilitar esto, C proporciona un archivo de encabezado, `inttypes.h`, que especifica explícitamente el ancho y el formato de cada tipo de datos. Estos tipos de datos son los siguientes:

- `int8_t`: entero de 8 bits con signo
- `uint8_t`: entero de 8 bits sin signo
- `int16_t`: entero de 16 bits con signo
- `uint16_t`: entero de 16 bits sin signo
- `int32_t`: entero de 32 bits con signo
- `uint32_t`: entero de 32 bits sin signo

- `int64_t`: entero de 64 bits con signo
- `uint63_t`: entero de 64 bits sin signo

Es una buena práctica usar estos tipos de datos para la codificación dedicada.

5.2.4.3 Chu_io_rw.h

La porción de código correspondiente al código para el acceso a E/S en la sección 5.2.3.2 (“Puntero C para registro de E/S”) es complejo y de difícil comprensión. Con algunas modificaciones se puede hacer que la operación sea más robusta y menos propensa a errores:

Primero, se puede agregar un tipo de conversión, (`volatile uint32_t *`).

```
sw = *(volatile uint32_t *)(0xc0000180);
```

La parte `uint32_t*` indica que el valor constante es un puntero que apunta a un objeto con el tipo de datos `uint32_t`. La palabra clave `volatile`, informa al compilador que el valor del objeto puede modificarse sin la interacción del procesador y no deben realizarse ciertas optimizaciones.

Segundo, se pueden usar palabras constantes simbólicas para la dirección base del núcleo GPI y el desplazamiento del registro de datos:

```
#define sw_base     0xc0000180
#define data_reg    0
. . .
sw = *(volatile uint32_t *)(sw_base + 4*data_reg);
```

En tercer lugar, para mantener la modularidad y mejorar la capacidad de ejecución, se puede definir una macro, `io_read()`, para encapsular las operaciones de conversión y desreferencia de tipos:

```
#define io_read (base_addr, offset) \
        (*(volatile uint32_t *)((base_addr) + 4*(offset)))
```

La declaración anterior se convierte en:

```
#define sw_base     0xc0000180
#define data_reg    0
. . .
sw = io_read (sw_base, data_reg);
```

También se define una macro auxiliar, `get_slot_addr()`, para calcular la dirección base de una ranura determinada y permite hacer referencia a un bloque

de E/S con la constante de ranura simbólica definida en `chu_io_map.h`:

```
#define get_slot_addr(base, slot) \
       ((uint32_t)((base) + (slot)*32*4))
#include  “chu_io_map.h”
. . .
#define sw_base   get_slot_addr(BRIDGE_BASE, S3_SW)
#define data_reg  0
. . .
sw = io_read (sw_base, data_reg);
```

Una macro similar, `io_write()`, se crea de manera similar:

```
#define io_write(base_addr, offset, data) \
 (*(volatile uint32_t *)((base_addr) + 4*(offset)) = (data))
```

Y se puede actualizar como:

```
#include  “chu_io_map.h”
. . .
#define led_base   get_slot_addr (BRIDGE_BASE, S2_LED)
#define  data_reg  0
. . .
Io_write (led_base, data_reg, pattern)
```

Estas macros se definen en `chu_io_wr.h` y el segmento de código se muestra en el Listado 2.2.

Listado 2.2. Macros de E/S (`chu_io_rw.h`)

```
#ifndef _CHU_IO_RW_H_INCLUDED
#define _CHU_IO_RW_H_INCLUDED

// library
#include <inttypes.h>    // to use unitN_t type
#ifdef __cplusplus
extern "C" {
#endif
#define io_read(base_addr, offset) \
   (*(volatile uint32_t *)((base_addr) + 4*(offset)))

#define io_write(base_addr, offset, data) \
   (*(volatile uint32_t *)((base_addr) + 4*(offset)) = (data))

#define get_slot_addr(base, slot) \
      ((uint32_t)((base) + (slot)*32*4))

#ifdef __cplusplus
} // extern "C"
#endif
#endif /* _CHU_IO_RW_H_INCLUDED */
```

Las macros pueden sufrir algún problema cuando un procesador contiene un caché de datos. Un caché de datos es un búfer rápido entre el procesador y la

memoria principal lenta que almacena los datos utilizados recientemente para reducir la necesidad de que el procesador acceda a la memoria externa lenta. Los datos relevantes pueden almacenarse temporalmente en el caché de datos y solo pueden leerse o escribirse en el registro de E/S cuando el bloque correspondiente se desasigna del caché. Para evitar esto, la rutina de arranque generalmente incluye un segmento de código para indicar las regiones de memoria que no son “cacheables”.

Esto no es un problema para MicroBlaze MCS ya que no tiene caché de datos.

5.2.5 TÉCNICAS PARA OPERACIONES DE E/S DEDICADAS

Debido a que un programa en estos sistemas interactúa con dispositivos de E/S dedicados, con frecuencia necesita manipular uno o más bits o un campo de objeto de datos.

5.2.5.1 Manipulación de bits

El lenguaje C tiene varios operadores bit a bit, que incluyen ~ (not), & (and), | (or) y ^ (xor), que operan en uno o dos operandos a nivel de bits.

El operador ~ invierte todos los bits individuales.

Los operadores &, | y ^ se pueden usar para manipular un bit o un grupo de bits en un objeto de datos. La operación implica un operando de datos y un operando de máscara, que especifica los bits a modificar.

La operación de conmutación se basa en la observación de que para cualquier variable booleana de 1 bit x, x^0= x y x^1=x’.

Muchas manipulaciones se realizan con operaciones de un solo bit. Una forma fácil de crear una máscara para el bit `n` es desplazando 1 hacia la izquierda para `n` posiciones, como en

```
1 UL << (n)
```

El UL significa que 1 está en formato largo sin signo y se incluye para evitar el desbordamiento.

Un conjunto de macros para realizar operaciones sobre un único bit se muestran a continuación:

```
#define bit_set(data, n) ((data) |= (1UL << (n)))     //pone a 1 el bit enésimo
#define bit_clear(data, n) ((data) &= ~(1UL << (n)))//pone a 0 el bit enésimo
#define bit_toggle(data, n) ((data) ^= (1UL << (n))) //conmuta el bit enésimo
#define bit_read(data, n) (((data) >> (n)) & 0x01) //lee estado del bit enésimo
#define  bit_write(data,  n,  bitvalue)  (bitvalue  ?  bit_set((data),  n)  :
```

```
bit_clear((data), n)) //pone a 1 o a 0 el bit enésimo de data
#define bit(n) (1UL << (n))
```

Estas macros se incluyen en el archivo `chu_init.h` de la Sección 5.2.7.2.

5.2.5.2 Empaquetado y desempaquetado

Para ahorrar espacio de direcciones, un registro de E/S contiene con frecuencia múltiples campos que se extraen y se desempaquetan después de que un programa de aplicación lee el registro de E/S. A la inversa, estos campos deben empaquetarse en un objeto cuando se escriben en el registro de E/S. Los procesos de desembalaje y empaquetado se pueden realizar mediante la manipulación a nivel de bits y la operación de cambio.

5.2.6 CONTROLADORES DE DISPOSITIVOS

5.2.6.1 Visión general

Un controlador de dispositivo es una de las rutinas posteriores que operan y controlan un dispositivo periférico en particular. En un sistema simple, un controlador actúa como una interfaz entre el hardware y un programa de aplicación y permite que el programa de aplicación acceda a las funciones de hardware sin necesidad de conocer detalles precisos. Debe ser “fácil de usar correctamente y difícil de usar incorrectamente”.

Se utiliza la clase C++ para el desarrollo del controlador debido a su soporte con la encapsulación de datos. Se define una clase para cada núcleo IP y contiene los siguientes componentes:

- *Inicialización de la rutina*. Este código está asociado con el constructor de la clase y se ejecutará cuando se cree en una instancia.
- *Operaciones*. Las operaciones principales son definidas e invocadas por las funciones *miembro* (*métodos*) de la clase.
- *Estado del dispositivo*. La información se mantiene en la sección privada de la clase y solo se puede acceder a ella mediante los métodos definidos en la clase.

Una clase puede considerarse como una unidad de software independiente “autónoma” asociada con un núcleo de E/S específico.

El programa de aplicación crea una instancia de clase para cada núcleo de IP instanciado al principio del código. El proceso realiza la inicialización en el núcleo y reserva espacio para su estado interno. A continuación, se muestran la

construcción de controladores para los núcleos GPI, GPO y temporizador y describen las funciones del núcleo de UART.

5.2.6.2 Controladores GPO y GPI

Es una buena práctica dividir una clase de C++ en dos archivos. La definición de la clase va al archivo de encabezado (con una extensión de `.h`) y la implementación de clase va al archivo de código C++ (con una extensión de `.cpp`).

Se usa una clase para cada núcleo. La definición de clase de del núcleo GPO se muestra en el Listado 2.3, donde se usa la convención de nomenclatura en la que el nombre del a clase comienza con una letra mayúscula y tiene una letra mayúscula para cada palabra nueva.

Listado 2.3. Definición de la clase GpoCore `(gpio_core.h)`

```
class GpoCore {
/* register map */
   enum {
      DATA_REG = 0 //data register
   };
public:
   GpoCore(uint32_t core_base_addr);       //constructor
   ~GpoCore();                             //destructor; not used
/* methods */
   void write(uint32_t data);              //write a 32-bit word
   void write(int bit_value, int bit_pos); //write 1 bit
private:
   uint32_t base_addr;
   uint32_t wr_data;                       // same as GPO core data reg
};
```

El mapa de registro de E/S del núcleo se lista primero con una declaración de enumeración. El núcleo de GPO solo tiene un registro de salida de datos, `DATA_REG`, cuyo desplazamiento es 0.

La siguiente parte es la sección pública, que contiene los métodos constructor, destructor y público. Hay dos funciones de escritura sobrecargadas. El primer método escribe una palabra de datos completa en el registro de datos y el segundo método escribe un único bit en una posición específica en el registro de datos. El destructor generalmente no es necesario para nuestros propósitos, pero se incluye para completar.

La última es la sección privada, que contiene dos variables. La variable `base_addr` almacena la dirección base del núcleo instanciado. La variable `wr_data` mantiene una copia de los datos escritos en el núcleo de GPO y puede considerarse como el *estado* del núcleo. Se vuelve útil cuando solo se actualiza parte del registro de datos.

La implementación de clase del núcleo GPO se muestra en el Listado 2.4:

Listado 2.4. Implementación de la clase GpoCore (gpio_core.cpp)

```
GpoCore::GpoCore(uint32_t core_base_addr) {
   base_addr = core_base_addr;
   wr_data = 0;
}
GpoCore::~GpoCore() {}

void GpoCore::write(uint32_t data) {
   wr_data = data;
   io_write(base_addr, DATA_REG, wr_data);
}
void GpoCore::write(int bit_value, int bit_pos) {
   bit_write(wr_data, bit_pos, bit_value);
   io_write(base_addr, DATA_REG, wr_data);
}
```

El constructor almacena la dirección base en la sección privada y borra los datos de escritura. El primer método `write()` actualiza el `wr_data` y luego escribe la palabra en el registro de datos del núcleo de GPO. El segundo método `write()` actualiza solo un bit en el registro de datos del núcleo. Hay que tener en cuenta que el núcleo de GPO está diseñado para aceptar toda la palabra. La modificación de un solo bit se logra con el controlador de software. La función `bit_write()` actualiza el bit designado en `wr_data` pero mantiene otros bits intactos. La palabra `wr_data` completa se escribe en el registro de datos del núcleo de GPO.

La definición de clase y la implementación del núcleo GPI se muestran en los listados 2.5 y 2.6. El núcleo GPI solo tiene un registro de entrada de dato, cuyo desplazamiento es 0. Son similares a los del núcleo GPO. Sin embargo, no es necesario mantener una copia del registro de datos de GPI, ya que se puede recuperar en cualquier momento.

Listado 2.5. Definición de la clase GpiCore (gpio_core.h)

```
class GpiCore {
   /* register map */
   enum {
      DATA_REG = 0 /**< input data register */
   };
public:
   GpiCore(uint32_t core_base_addr);         //constructor
   ~GpiCore();                               //destructor; not used
   /* methods */
   uint32_t read();                          //read a 32-bit word
   int read(int bit_pos);                    //read 1 bit
private:
   uint32_t base_addr;
};
```

Listado 2.6. Implementación de la clase GpiCore (gpio_core.cpp)

```
GpiCore::GpiCore(uint32_t core_base_addr) {
   base_addr = core_base_addr;
}
GpiCore::~GpiCore() {
}
uint32_t GpiCore::read() {
   return (io_read(base_addr, DATA_REG));
}
```

```
int GpiCore::read(int bit_pos) {
   uint32_t rd_data = io_read(base_addr, DATA_REG);
   return ((int) bit_read(rd_data, bit_pos));
   }
```

En el sistema FPro, todos los registros del núcleo de E/S se tratan como palabras de 32 bits. Los anchos reales pueden variar. En una implementación más robusta, este tipo de información se debe pasar desde la configuración del hardware al controlador del software. Los métodos deben comprobar errores o enmascarar los bits no utilizados. Sin embargo, este tipo de integración introducirá una cantidad significativa de sobrecarga y no se implementa en el marco de FPro. Tenemos que ser conscientes de la situación que ocurre y manejarla manualmente en el programa de aplicación.

5.2.6.3 Controlador del temporizador

La interfaz principal del temporizador IP utiliza tres registros y su mapa de E/S se muestra en la Figura 2.3. El bloque contiene un contador de gran tamaño (64 bits). Se ejecuta de forma continua, pero puede pausarse o borrarse mediante las señales de control. Los registros 0 y 1 corresponden a los 32 LSB y 32 MSB del contador. Los bits 0 y 1 del registro 3 están conectados a las señales de borrado y habilitación de cuenta. Escribir un 0 en el bit 0 detendrá el conteo y escribir un 1 reanudará el conteo. El bit 1 no es realmente en un elemento de memoria. Escribir un 1 en el bit 1 genera un reset que limpia el contador a 0. La definición de clase del núcleo del temporizado se muestra en el Listado 2.7.

Listado 2.7. Definición de la clase TimerCore `(timer_core.h)`

```
class TimerCore {
   */ register map */
   enum {
      COUNTER_LOWER_REG = 0, /**< lower 32 bits of counter */
      COUNTER_UPPER_REG = 1, /**< upper 16 bits of counter */
      CTRL_REG = 2           /**< control register */
   };
   /* masks */
   enum {
      GO_FIELD = 0x00000001, /**< bit 0 of ctrl_reg; enable bit  */
      CLR_FIELD = 0x00000002 /**< bit 1 of ctrl_reg; clear bit */
   };
public:
   TimerCore(uint32_t core_base_addr);    //constructor
   ~TimerCore();                          //destructor; not used
   /* methods */
   void pause();                         //pause counter
   void go();                            //resume counter
   void clear();                         //clear the counter to 0
   uint64_t read_tick();                 //retrieve # clocks elapsed
   uint64_t read_time();                    //read time elapsed (in microsecond)
   void sleep(uint64_t us);              //idle for us microseconds
private:
   uint32_t base_addr;
   uint32_t ctrl;                         // current state of control register
   };
```

La estructura básica es similar a la del `GpoCore`. La primera definición de enumeración utiliza nombres simbólicos para las tres posiciones de registro. Se agrega otra definición para especificar las máscaras para extraer y borrar bits. Los métodos se utilizan para controlar el contador, para extraer el valor actual del contador y para implementar una función de “suspensión”. La sección privada mantiene la dirección base y contiene una variable, `ctrl`, que mantiene una copia de los datos idéntica al contenido del registro de control del núcleo del temporizador. La implementación de clase del núcleo del temporizador se muestra en el Listado 2.8.

Listado 2.8. Implementación de la clase TimerCore `(timer_core.cpp)`

```
TimerCore::TimerCore(uint32_t core_base_addr) {
   base_addr = core_base_addr;
   ctrl = 0x01;
   io_write(base_addr, CTRL_REG, ctrl);  // enable the timer
}

TimerCore::~TimerCore() {}

void TimerCore::pause() {
   // reset enable bit to 0
   ctrl = ctrl & ~GO_FIELD;
   io_write(base_addr, CTRL_REG, ctrl);
}

void TimerCore::go() {
   // set enable bit to 1
   ctrl = ctrl | GO_FIELD;
   io_write(base_addr, CTRL_REG, ctrl);
}
void TimerCore::clear() {
   uint32_t wdata;

   // write clear_bit to generate a 1-clock pulse
   // clear bit does not affect ctrl
   wdata = ctrl | CLR_FIELD;
   io_write(base_addr, CTRL_REG, wdata);
}

uint64_t TimerCore::read_tick() {
   uint64_t upper, lower;

   lower = (uint64_t) io_read(base_addr, COUNTER_LOWER_REG);
   upper = (uint64_t) io_read(base_addr, COUNTER_UPPER_REG);
   return ((upper << 32) | lower);
}

uint64_t TimerCore::read_time() {
   // elapsed time in microsecond (SYS_CLK_FREQ in MHz)
   return (read_tick() / SYS_CLK_FREQ);
}

void TimerCore::sleep(uint64_t us) {
   uint64_t start_time, now;

   start_time = read_time();
   // busy waiting
   do {
      now = read_time();
   } while ((now - start_time) < us);
   }
```

El constructor almacena la dirección base e inicia el contador. El método `pause()` limpia el bit y detiene la cuenta. El método `go()` restablece el bit de habilitación de cuenta para permitir el conteo. El método `clear()` escribe el bit de borrado del registro de control. Hay que tener en cuenta que la escritura se realiza mediante la creación de una variable `wdata` sin almacenar el valor en ctrl.

El método `read_tick()` recupera el contenido de dos registros y los empaqueta para obtener el número de tics de reloj transcurridos desde el último borrado. El reloj marca el tiempo transcurrido en microsegundos. El método `sleep()` consiste en un bucle de espera activa. Comprueba el tiempo continuamente en microsegundos.

5.2.6.4 Controlador UART

El bloque de UART permite establecer un canal de comunicación serie, por ejemplo, con un computador a través del puerto serie. Su interfaz utiliza buffers FIFO en lugar de registros de E/S. Por lo tanto, la construcción de su clase de controlador, `UartCore`, es algo diferente. Simplemente se proporciona una descripción general de los métodos de alto nivel que se utilizarán más adelante. Estos métodos establecen la velocidad de transmisión en baudios y muestran una cadena o número simple:

- `Set_baud_rate(int baud)` establece la velocidad en baudios del UART. El constructor establece la velocidad de transmisión predeterminada en 9600 baudios. El método puede invocarse si se desea una velocidad de transmisión diferente.

- `Disp(const char * str)` transmite una cadena de caracteres, que se puede visualizar en un terminal o consola.

- `Disp(int n, int base, int len)` convierte el número n en una cadena y transmite la cadena. El parámetro `base` puede ser 2, 8, 10 o 16 y el parámetro `len` especifica el número de dígitos (longitud de la cadena) que se mostrará.

- `Disp(int n, int base)` es una versión sobrecargada en la que la longitud no se especifica, se determina automáticamente.

- `Disp(int n)` es otra versión sobrecargada en la que la longitud se determina automáticamente y se usa la base 10 (decimal).

- `Disp(double f, int digit)` convierte un numero de coma flotante `f` en una cadena y transmite la cadena.

- `Disp(double f)` es una versión sobrecargada en la que `digit` se establece por defecto en valor 3.

El método `disp()` se puede utilizar como una versión primitiva de `printf()`.

5.2.7 RUTINAS DE UTILIDAD DE FPRO Y ESTRUCTURA DEL DIRECTORIO

Un sistema dedicado se ejecuta en un entorno C independiente y no puede acceder a las bibliotecas C normales. Para facilitar el desarrollo del software, se desarrollan un conjunto de rutinas de utilidad simples para acceder por ejemplo, a la hora del sistema y comunicarnos con una consola. La definición y la implementación de estas rutinas se encuentran en `chu_init.h` y `chu_init.cpp`, respectivamente.

5.2.7.1 Requisitos mínimos de hardware

Las rutinas de utilidad imponen los siguientes requisitos:

- Un bloque de IP de Temporizador en la ranura 0 del subsistema MMIO.
- Un bloque UART IP en la ranura 1 del subsistema MMIO.

Por lo tanto, cuando se crea un sistema FPro, las primeras dos ranuras están reservadas y deben conectarse al Temporizador y a la UART. También se recomienda usar un núcleo de GPO para la ranura 2 y el núcleo de GPI para la ranura 3. Se pueden agregar núcleos IP adicionales de aceleradores de hardware y periféricos al resto de las ranuras del sistema para satisfacer una necesidad específica.

5.2.7.2 Rutinas de utilidad

Hay cuatro tipos de rutinas de utilidad:

- Funciones de visualización de la consola
- Funciones de tiempo
- Depuración de mensajes
- Macros de manipulación de bits

Los archivos de definición e implementación se muestran en los Listados 2.9 y 2.10.

Listado 2.9. Declaraciones y macros de las rutinas FPro `(chu_init.h)`

```
// library
#include "chu_io_rw.h"
#include "chu_io_map.h"
#include "timer_core.h"
```

```
#include "uart_core.h"

//  make uart visible by other code
extern UartCore uart;

// define timer and uart slots
#define TIMER_SLOT 0
#define UART_SLOT 1

//timing functions
unsigned long now_us();
unsigned long now_ms();
void sleep_us(unsigned long int t);
void sleep_ms(unsigned long int t);

//define debug function
void debug_off();
void debug_on(const char *str, int n1, int n2);

#ifndef _DEBUG
#define debug(str, n1, n2) debug_off()
#endif // not _DEBUG

#ifdef _DEBUG
#define debug(str, n1, n2) debug_on((str), (n1), (n2))
#endif // not _DEBUG

//low-level bit-manipulation macros
#define bit_set(data, n) ((data) |= (1UL << (n)))
#define bit_clear(data, n) ((data) &= ~(1UL << (n)))
#define bit_toggle(data, n) ((data) ^= (1UL << (n)))
#define bit_read(data, n) (((data) >> (n)) & 0x01)
#define bit_write(data, n, bitvalue) (bitvalue ? bit_set((data), n) : bit_clear((data),
   n))
#define bit(n) (1UL << (n))

#endif  // _CHU_INIT_H_INCLUDED
```

Listado 2.10. Implementación rutinas de utilidad FPro (chu_init.cpp)

```
TimerCore _sys_timer(get_slot_addr(BRIDGE_BASE, TIMER_SLOT));
UartCore uart(get_slot_addr(BRIDGE_BASE, UART_SLOT));

// current system time in microsecond
unsigned long now_us() {
   return ((unsigned long) _sys_timer.read_time());
}

// current system time in ms
unsigned long now_ms() {
   return ((unsigned long) _sys_timer.read_time() / 1000);
}

// idle for t microseconds
void sleep_us(unsigned long int t) {
   _sys_timer.sleep(uint64_t(t));
}

// idle for t ms
void sleep_ms(unsigned long int t) {
   _sys_timer.sleep(uint64_t(1000 * t));
}

// uart print a 1-line message: msg + 2 numbers in dec/hex format
void debug_on(const char *str, int n1, int n2) {
   uart.disp("debug: ");
   uart.disp(str);
   uart.disp(n1);
   uart.disp("(0x");
   uart.disp(n1, 16);
   uart.disp(") / ");
```

```
    uart.disp(n2);
    uart.disp("(0x");
    uart.disp(n2, 16);
    uart.disp(") \n\r");
}

void debug_off() {
}
```

Funciones de visualización de la consola. Una instancia, llamada `uart`, con la clase `UartCore` se crea en `chu_init.cpp` y se hace visible a todos los códigos externos. Por lo tanto, la instancia y los métodos de la clase `UartCore` se pueden usar en códigos externos para mostrar una cadena o un número en la consola:

- `Uart.disp (const char * str)`
- `Uart.disp (int n, int base, int len)`
- `Uart.disp (int n, int base)`
- `Uart.disp (int n)`
- `Uart.disp (double f, int digit)`
- `Uart.disp (double f)`

Funciones de temporización. Se basan en el núcleo del temporizador creado en la ranura 0. Una instancia con la clase `TimerCore`, `_sys_timer`, se crea en `chu_init.cpp`. La instancia está configurada para ejecutarse continuamente desde el inicio y esencialmente mantiene el tiempo de funcionamiento del sistema. No es visible para archivos externos y el temporizador no puede pausarse ni borrarse. Varias funciones de utilidad se derivan del controlador de temporizador:

- `Unsigned long now_us()`: devuelve la hora del sistema en microsegundos
- `Unsigned long now_ms()`: devuelve la hora del sistema en milisegundos
- `Void sleep_us (unsigned long int t)`: obliga al sistema a inactivarse (espera activa) por `t` microsegundos
- `Void sleep_ms (unsigned long int t)`: obliga al sistema a inactivarse (espera activa) por `t` milisegundos

Depuración de mensajes. Una función de depuración, `debug()`, combinada con la directriz `_DEBUG`, proporciona una forma sencilla de facilitar el proceso de desarrollo de software. Dos versiones de `debug()`, que son `debug_off()` y

`debug_on()`, están definidas en el archivo de encabezado. Según el estado de la directriz `_DEBUG`, una se usa durante el preprocesamiento. La función `debug_off()` no contiene ningún código y la función `debug_on()` muestra una línea en un formato fijo, que consiste en una cadena, y dos números en formato decimal y hexadecimal.

Se puede activar la mensajería de depuración durante el desarrollo inicial del código agregando una instrucción directiva en el programa principal

```
#define _DEBUG
```

Cuando la ejecución del programa llegue a esta línea, se mostrará un mensaje de depuración en la consola:

```
debug: timer check – (loop #)/now: 2(0x02) / 386735)0x5e6af)
```

Podemos desactivar el mensaje de depuración más tarde simplemente eliminando o comentando la directriz _DEBUG.

Macros de manipulación de bits. Un conjunto de macros se construyen para realizar operaciones de un solo bit:

- `Bit_set (datos, n)`: establece el bit `n` de `datos` en 1.
- `Bit_clear (datos, n)`: borra el bit `n` de `datos` a 0.
- `Bit_toggle (datos, n)`: alterna el bit `n` de los `datos`.
- `Bit_read (datos, n)`: devuelve el valor del bit `n` de los `datos`.
- `Bit_write (datos, n)`: actualiza el bit `n` de los `datos` con un valor bitvalue.
- `Bit(n)`: devuelve una máscara con el bit `n` activado.

5.2.7.3 Estructura de directorio

La jerarquía de archivos de encabezado se muestra en la Ilustración 10. El archivo `chu_init.h` crea una instancia de un núcleo de temporizador y un `uart`, por lo que necesita los archivos de `timer_core.h` y `uart_core.h`. Las clases de `TimerCore` y `UartCore` utilizan `chu_io_map.h` y `chu_io_rw` para las operaciones básicas de E/S. Las otras clases principales de IP solo necesitan incluir `chu_init.h`, que a su vez invoca los archivos de E/S básicos. El programa principal debe incluir los archivos de encabezado de los núcleos IP instanciados y el archivo `chu_init.h` según sea necesario.

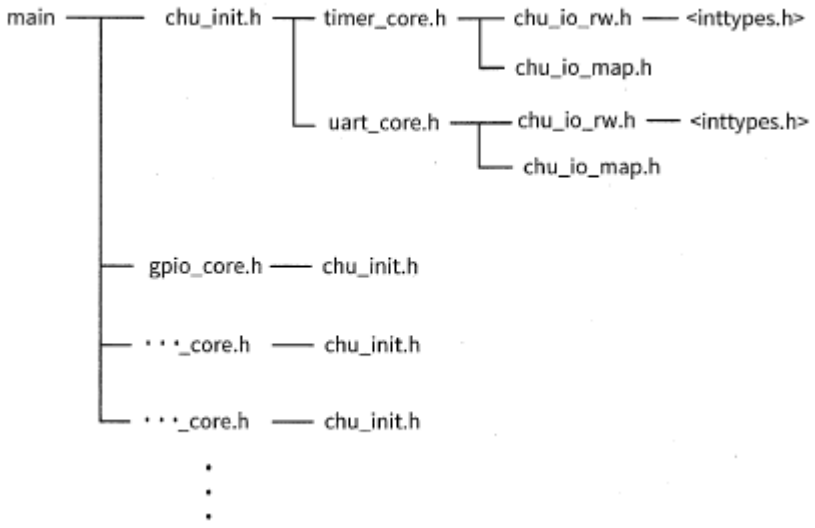


Ilustración 10: *Jerarquía de archivos.*

5.2.8 PROGRAMA DE PRUEBAS

5.2.8.1 Rutina de verificación de núcleo IP

Después de que se desarrollen un hardware y un controlador de IP, se necesita desarrollar una función de prueba de software para verificar su funcionamiento que debe ejercer todas las características del hardware y del controlador y prestar especial atención a las condiciones especiales y los “casos límite”.

5.2.8.2 Programación con memoria limitada

Como MicroBlaze MCS proporciona una memoria bastante limitada (hasta 128 MB), debemos tener en cuenta el tamaño del programa.

Después de compilar y vincular un programa de aplicación, su imagen contiene cuatro segmentos principales: texto, datos/bss, pila y memoria de pila. El segmento de texto es el código del programa y los datos/bss contienen datos globales (externos) y estáticos. Los tamaños de estos dos segmentos son fijos y conocidos. El segmento de pila almacena los parámetros y las variables locales para las llamadas de función y el segmento de pila proporciona espacio para los datos asignados dinámicamente. Los segmentos de pila y memoria de pila crecen y se reducen dinámicamente durante la ejecución. El programa se bloquea cunado se queda sin memoria.

Crear una instancia puede requerir un gran bloque de memoria y puede llevar a consumir demasiados recursos. Para aliviar el problema, adoptamos los siguientes enfoques para el sistema FPro:

- Crear una instancia externa (global) para cada núcleo físico. Esto permite que el objeto se asigne en el segmento de datos/bss y que todas las rutinas puedan acceder a él.
- Evitar el uso excesivo de clases e instancias de C++ e intentar usar construcciones C para las funcionalidades que no son controladores.
- Reducir el tamaño y la complejidad del controlador del dispositivo.

5.2.8.3 Integración de la función de prueba

La incorporación de la función de prueba en un programa principal implica los siguientes pasos:

- Incluir el archivo de encabezado del controlador central.
- Desarrollar la función de prueba.
- Crear una instancia externa.
- Llamar a la función en `main()`.

5.2.8.4 Programa de prueba para el sistema FPro

Se puede desarrollar un programa de prueba para verificar el funcionamiento de los cuatro bloques IP en el sistema de vanilla FPro, como se muestra en el Listado 2.11. Hay que tener en cuenta que el bloque de UART y el del Temporizador se usan implícitamente en el archivo `chu_init.h`, y por tanto, no se muestran en el programa principal.

Listado 2.11. Programa de prueba Vanilla FPro `(main_vanilla_test.cpp)`

```
//#define _DEBUG
#include "chu_init.h"
#include "gpio_cores.h"

void timer_check(GpoCore *led_p) {
   int i;

   for (i = 0; i < 5; i++) {
      led_p->write(0xffff);
      sleep_ms(500);
      led_p->write(0x0000);
      sleep_ms(500);
      debug("timer check - (loop #)/now: ", i, now_ms());
   }
}

void led_check(GpoCore *led_p, int n) {
   int i;

   for (i = 0; i < n; i++) {
      led_p->write(1, i);
      sleep_ms(200);
      led_p->write(0, i);
      sleep_ms(200);
   }
}
```

```
void sw_check(GpoCore *led_p, GpiCore *sw_p) {
   int i, s;

   s = sw_p->read();
   for (i = 0; i < 30; i++) {
      led_p->write(s);
      sleep_ms(50);
      led_p->write(0);
      sleep_ms(50);
   }
}

void uart_check() {
   static int loop = 0;

   uart.disp("uart test #");
   uart.disp(loop);
   uart.disp("\n\r");
   loop++;
}

// instantiate switch, led
GpoCore led(get_slot_addr(BRIDGE_BASE, S2_LED));
GpiCore sw(get_slot_addr(BRIDGE_BASE, S3_SW));

int main() {

   while (1) {
      timer_check(&led);
      led_check(&led, 16);
      sw_check(&led, &sw);
      uart_check();
      debug("main - switch value / up time : ", sw.read(), now_ms());
   } //while
} //main
```

Se crean primero una instancia de clase `GpiCore`, `sw`, y una instancia de clase `GpoCore`, `led`. El programa principal sigue la estructura de súper bucle. Este contiene cuatro funciones de prueba, una para cada bloque o núcleo. La función `timer_check()` hace parpadear todos los LED una vez por segundo cinco veces, `led_check()` enciende y apaga un LED individual en secuencia, `sw_check()` hace parpadear un LED si la entrada del interruptor correspondiente es 1, y `uart_check()` envía un mensaje simple a la consola vía puerto serie.

Se incluye una declaración de `debug()` en `timer_check()` y el programa principal y un mensaje de estado se transmitirá si la directriz `_DEBUG` no se comenta. Hay que tener en cuenta que no hay una rutina de inicialización explícita.

5.2.8.5 Implementación

La parte más problemática del uso de MicroBlaze MCS es descargar el archivo *.elf*. La memoria RAM de MCS se implementa utilizando los módulos BRAM internos de la FPGA. Desde el punto de vista técnico, debería haber dos formas de descargar el archivo *.elf*:

- El procedimiento que llamaremos “normal” primero se descarga el archivo *.bit* a un dispositivo FPGA (construyendo “hardware”) y luego escriben los módulos BRAM de FPGA con el archivo *.elf* (cargando el software), como se muestra en los pasos 5 y 8 de la Ilustración 4.

- El procedimiento que denominaremos “fusionado” trata el archivo *.elf* como los “valores iniciales” con los que inicializar los módulos BRAM de FPGA, los combina en el archivo *.bit* y luego descarga el archivo *.bit* combinado en un dispositivo FPGA, como se muestra en los pasos 8 y 5 en la Ilustración 4.

Por el momento, solo el procedimiento de fusionado es funcional, lo que obliga a invocar a Vivado para regenerar el archivo bitstream (*.bit*), para cada vez que hay una variación o revisión del software.

5.3 PROTOCOLO DE BUS DE FPRO Y ESPECIFICACIÓN DE DIRECCIÓN MMIO

Como ya se ha indicado, el sistema FPro utiliza un bus interno síncrono y define una interfaz con una dirección base o slot fijo para los núcleos de E/S MMIO. La interfaz de bus proporciona un mecanismo flexible para que el procesador se comunique con los núcleos de E/S y una forma sencilla y organizada de incorporar más bloques personalizados en el sistema.

5.3.1 BUS FPRO

5.3.1.1 Descripción general del BUS

Un *bus* es una conjunto de conexiones que permite interconectar múltiples componentes para que intercambien información. Un sistema de bus dentro de un ordenador es un bus para que un procesador intercambie datos con módulos de memoria y dispositivos de E/S. El procesador funciona como maestro y los módulos de memoria y dispositivos de E/S son los *esclavos*.

Se puede pensar en él, como en una colección de cables compartidos por todos los componentes conectados. El diagrama especificado se muestra en la Ilustración 11. Los cables se pueden clasificar en tres grupos en función de su funcionalidad o el tipo de datos que viajan por ellos:

- Líneas de datos.

- Líneas de dirección.

- Líneas de control.

Las *líneas de datos* llevan los datos a transferir. El número de líneas de datos representa el ancho de los datos. Un bus interno usualmente tiene líneas separadas para leer datos y escribir datos. Las *líneas de dirección* identifican la ubicación de una transacción. Con *n* líneas de dirección, se pueden asignar hasta 2^n ubicaciones distintas. Los bits superiores de las líneas de dirección se utilizan para especificar una ubicación dentro del módulo o dispositivo. Las *líneas de control* consisten en señales de comando y estado para controlar la transacción deseada.

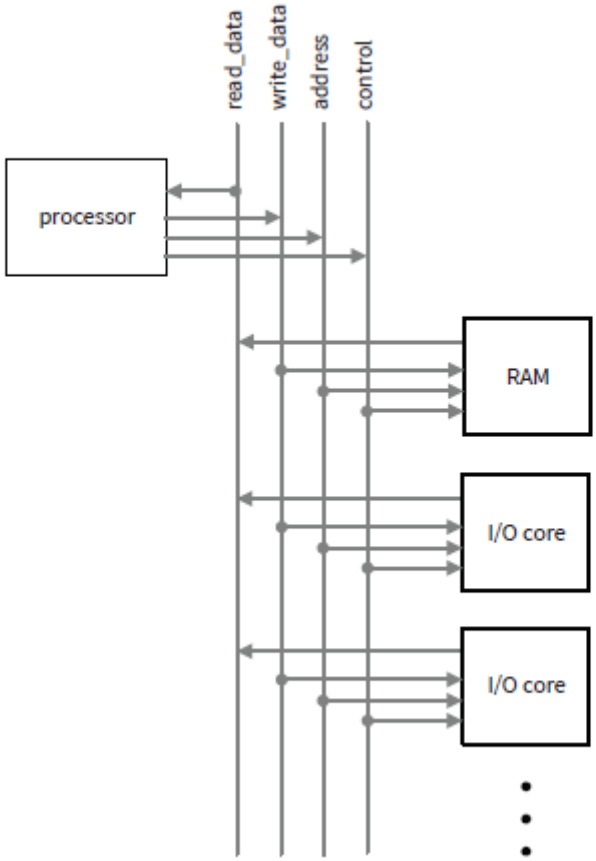


Ilustración 11: *Diagrama de bus conceptual.*

5.3.1.2 Interconexión SoC

A medida que crecen las funcionalidades de un sistema, la arquitectura del bus se vuelve más compleja y tiende a convertirse en un cuello de botella para el sistema, ya que no es posible que cohexista en el bus más que un único dato a la vez. Se agrava en el desarrollo de SoC, ya que contiene múltiples subsistemas que interactúan y se comunican entre sí, y deben compartir por turnos, las líneas físicas de interconexión. Como alternativa, en lugar de utilizar un único bus centralizado, se proporciona al sistema una *infraestructura de comunicación*

distribuida, a veces denominada *interconexión*. Los nuevos dispositivos FPGA de Xilinx adoptan como estándar el protocolo ARM ABA AXI para su infraestructura de comunicación.

Si bien el protocolo AXI es muy utilizado y flexible, es demasiado complejo para nuestros propósitos de comunicación. Además, aunque el protocolo AXI es un estándar abierto, su implementación se basa en los núcleos IP de interconexión del proveedor y, por lo tanto, puede introducir problemas de portabilidad. Definiremos por tanto, un protocolo de bus mucho más simple para el sistema FPro.

5.3.1.3 Especificación del protocolo del bus FPro

El bus FPro es el medio de comunicación compartido por los núcleos de E/S del MMIO y los subsistemas de video. El bus FPro es un bus de sistema síncrono simple que solo admite operaciones de lectura y escritura. El procesador es el maestro y puede iniciar una transacción de lectura o escritura a través del puente FPro. Los núcleos de E/S en dos subsistemas son esclavos.

Además del reloj y reinicio del sistema, el bus FPro contiene las siguientes señales:

- `fp_addr` (maestro a esclavo). Es una señal de dirección de 22 bits utilizada para identificar el registro de E/S de destino o una ubicación de memoria en el MMIO o subsistemas de video. Tenga en cuenta que el espacio de memoria del subsistema de E/S FPro es "direccionable por palabra", lo que significa que la ubicación especificada por `fp_addr` es una palabra de 32 bits, por lo tanto habría que completar la dirección de `fp_addr` de 22 bits anteponiendo diez ceros.

- `fp_rd_data` (esclavo a maestro). Es una señal de 32 bits que transporta los datos leídos.

- `fp_wr_data` (maestro a esclavo). Es una señal de 32 bits que lleva los datos a escribir.

- `fp_rd` (maestro a esclavo). Es una señal de control de 1 bit asociada con una operación de lectura.

- `fp_wr` (maestro a esclavo). Es una señal de control de 1 bit para iniciar una operación de escritura.

- `fp_mmio_cs` (maestro a esclavo). Es una señal de habilitación de 1 bit ("selección de chip") para activar el subsistema MMIO.

- `fp_video_cs` (maestro a esclavo). Es una señal de habilitación de 1 bit para activar el subsistema de video.

Las operaciones de lectura y escritura del bus FPro son sincronas y se deben completar en un ciclo de reloj. El diagrama de tiempo se muestra en la Ilustración 12. Se supone que la operación se realiza en el subsistema MMIO. El tiempo del sistema de video es similar, excepto si el `fp_video_cs` es usado.

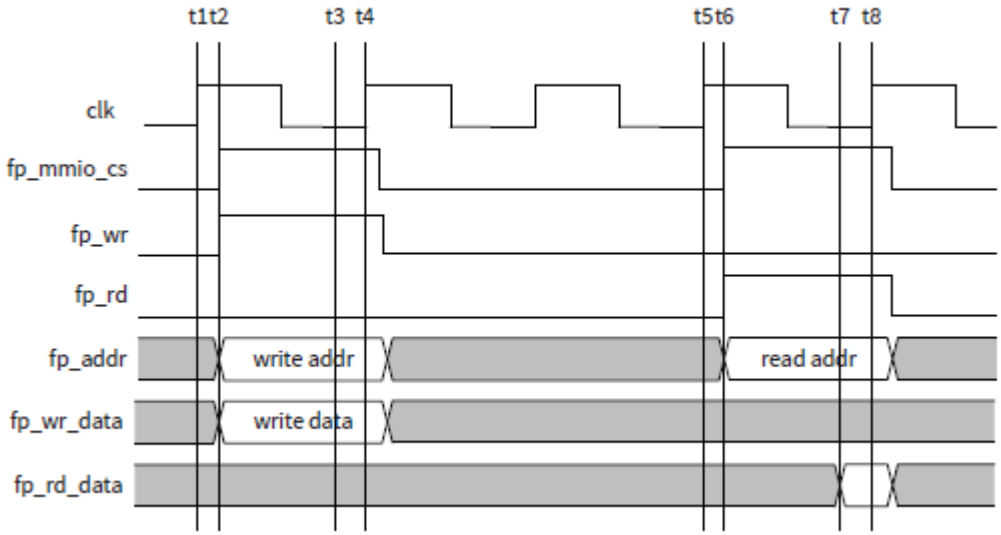


Ilustración 12: *Diagrama de tiempo del bus FPro.*

La parte izquierda muestra el ciclo de escritura. En la marca *t1*, el puente completa la traducción del comando de escritura desde el bus nativo del procesador, coloca la dirección y escribe los datos en `fp_addr` y `fp_we_data`, y activa las señales `dp_mmio_cs` y `fp_wr`. La lógica decodifica la dirección y activa la señal de habilitación del registro de E/S designado. En la marca *t4*, el registro designado muestrea y almacena los datos de escritura.

La parte derecha muestra el ciclo de lectura. En la marca *t5*, el puente completa la traducción del comando de lectura desde el bus nativo del procesador, coloca la dirección en `fp_addr` y activa las señales de `fp_mmio_cs` y `fp_rd`. La lógica de multiplexación examina las direcciones y enruta la salida del registro de E/S designada a `fp_rd_data`. En la marca *t8*, el puente muestrea y almacena los datos y los envía al procesador a través de su bus nativo. Tenga en cuenta que la señal `fp_rd` no participa directamente en la recuperación de datos de lectura.

En realidad, funciona como una señal de “eliminación” que elimina el elemento de datos antiguo en el núcleo de E/S para crear un lugar para uno nuevo.

La operación de un ciclo impone una restricción de tiempo estricta en los núcleos conectados al bus. Sin embargo, dado que estos núcleos están diseñados desde cero, se puede incorporar un buffer adecuado en su interfaz para satisfacer el requerimiento que impone la temporización.

5.3.2 INTERFAZ CON EL BUS

5.3.2.1 Introducción

Una tarea principal en el diseño del SoC es integrar la lógica personalizada en el sistema. Una forma común es adjuntar la lógica personalizada al bus del sistema y acceder a ella como bloques de E/S. Este proceso involucra dos partes:

- Agregar un *circuito que integre “envuelva”* a la lógica personalizada desde un bloque o núcleo de E/S compatible que pueda entrar en interfaz con el bus del sistema.

- Actualizar el *circuito de decodificación y multiplexación a nivel de sistema* para identificar y poder acceder al núcleo.

El *circuito envolvente* hace que el núcleo se parezca a una pequeña memoria y se pueda direccionar y acceder en consecuencia. Contiene una colección de registros, un *circuito de decodificación* y un *circuito de multiplexación*. El circuito de decodificación decodifica las líneas de dirección para identificar y habilitar el registro de destino designado. El circuito de multiplexación utiliza las líneas de dirección para seleccionar el registro de fuente designado y encamina los datos al bus.

Las funcionalidades de un circuito de decodificación a nivel de sistema y un circuito de multiplexación son similares a las del núcleo de E/S, excepto que se usan para identificar un núcleo de E/S instanciado en lugar de una única ubicación de registro.

En el esquema de mapeado de memoria de E/S, el sistema asigna un pequeño espacio de memoria consecutivo a cada núcleo de E/S. Esto implica que cada núcleo está asociado con un rango específico del espacio de direcciones. Desde esta perspectiva, podemos dividir las líneas de dirección en bits de módulo y bits de desplazamiento. Los bits de desplazamiento constituyen la parte inferior de la dirección y su ancho es idéntico al ancho de las líneas de dirección del núcleo de E/S. Los bits del módulo constituyen la parte restante de la dirección. Hay que tener en cuenta que los bits de desplazamiento de la dirección deben ser ceros.

La implementación real del circuito de envoltura del bloque de E/S de FPro y el controlador MMIO se expone en las secciones 5.3.3 y 5.3.5, respectivamente.

5.3.2.2 Interfaz de escritura y decodificación

Durante una operación de escritura, el maestro coloca los datos y la dirección de destino en el bus y activa la señal de escritura. Los datos se transmiten a través del bus de datos a todos los módulos de memoria y registros de E/S. Una secuencia de circuitos de decodificación toma la dirección y sus salidas afirman las señales de habilitación del núcleo de E/S designado y el registro designado para que los datos se almacenen en este registro en particular.

El diagrama de bloques del circuito de interfaz de escritura se muestra en la Ilustración 13. Hay que tener en cuenta que el núcleo de E/S tiene una señal `cs` de un bit (para "selección de chip") para seleccionar y habilitar el bloque.

Escritura de la interfaz del circuito de envoltura. Las partes clave del circuito de envoltura del núcleo de E/S son cuatro registros y un circuito de decodificación. La tabla de funciones del circuito de decodificación se muestra en la Tabla 1. Es básicamente un decodificador de 2 a 2^2. Las entradas del decodificador están conectadas a los dos LSB de las líneas de dirección del bus. Si se selecciona el núcleo (`cs` es 1), la salida descodificada habilita el registro correspondiente y los datos de escritura se muestrean y almacenan en el registro en el flanco ascendente del reloj.

Circuito de decodificación a nivel de sistema. La decodificación a nivel del sistema está vinculada a la dirección base del núcleo. Supongamos que se asigna un bloque de cuatro palabras con una dirección base de 100011_00 a este módulo instanciado. El circuito de decodificación consiste en un comparador "igual a" que compara los bits del módulo (6 MSBs) de las líneas de dirección con 100011. Cuando el valor coincide, afirma `cs` del bloque de E/S y activa ese bloque.

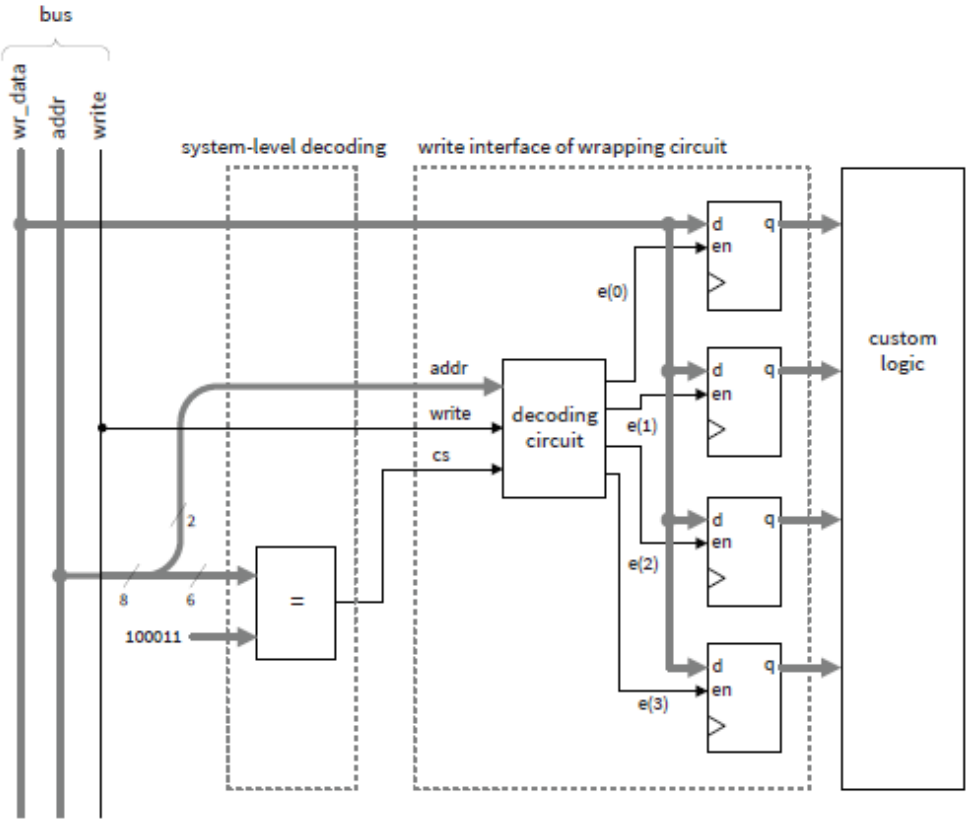


Ilustración 13: *Diagrama de bloques de una interfaz de escritura.*

input			output
cs	write	addr	e
0	–	–	0000
1	0	–	0000
1	1	00	0001
1	1	01	0010
1	1	10	0100
1	1	11	1000

Tabla 1: *Tabla de funciones de un circuito de decodificación.*

5.3.2.3 Interfaz de lectura y multiplexación

Durante una operación de lectura, el maestro coloca la dirección de destino en el bus y activa la señal de lectura. Una secuencia de *circuitos de multiplexación* selecciona y enruta los datos desde el registro fuente designado y los coloca en las líneas de datos de lectura del bus. El maestro luego recupera los datos de lectura. El diagrama de bloques del circuito de interfaz de lectura se muestra en la Ilustración 14.

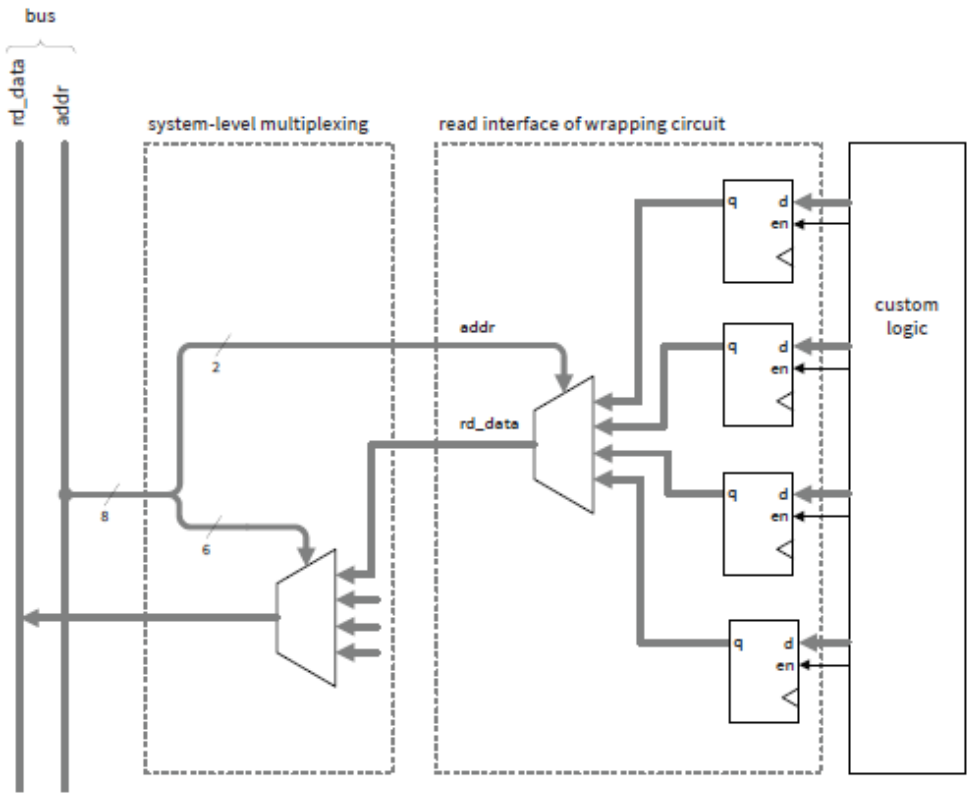


Ilustración 14: *Diagrama de bloques de una interfaz de lectura.*

Interfaz de lectura del circuito de envoltura. Las partes clave del circuito de envoltura del bloque son cuatro registros y un circuito de multiplexación. El circuito de multiplexación es un multiplexor estándar de 2^2 a 1. Los LSB más bajos de las líneas de dirección se utilizan para identificar la fuente. Se conectan mediante la señal de selección del multiplexor para encaminar la fuente designada al circuito de multiplexación a nivel del sistema. Tenga en cuenta que la señal de lectura no se utiliza en esta configuración.

Circuito de multiplexación a nivel de sistema. Constituye el segundo nivel de enrutamiento y contiene otro multiplexor. Sus puertos de entrada están conectados a los datos de lectura de varios núcleos de E/S y sus datos de salida están conectados a las líneas de datos de lectura del bus. Los seis bits del módulo se utilizan como la señal de selección y enrutan los datos leídos desde el módulo designado al procesador.

5.3.2.4 Búfer FIFO como un registro de E/S

Un registro de E/S de un núcleo se trata como una ubicación de memoria y el procesador lee y escribe el registro directamente. Sin embargo, existe una diferencia entre una ubicación de memoria normal y un registro de E/S. Para una ubicación de memoria, el procesador realiza operaciones de lectura y escritura

y, por lo tanto, siempre conoce la "validez" de los datos de esta ubicación. Para un registro de E/S, en contraste, el procesador solo es responsable de la "mitad" del acceso. En una operación de escritura, el procesador escribe (es decir, produce) los datos y un núcleo de E/S lee (es decir, consume) los datos. En una operación de lectura, el procesador lee (es decir, consume) los datos y un núcleo E/S escribe (es decir, produce) los datos.

Como un procesador generalmente funciona más rápido que los núcleos de E/S, la tasa de producción y la tasa de consumo pueden no ser las mismas y los errores, como leer los mismos datos recibidos desde un núcleo UART lento varias veces, pueden ocurrir en el acceso a los datos. Una forma de resolver el problema es reemplazar un registro de E/S con un búfer FIFO más general. El diagrama de bloques de la interfaz de escritura con esta modificación se muestra en la Ilustración 15. Las señales de habilitación decodificadas están conectadas a las señales de escritura (`wr`) de las FIFO. Un procesador puede transmitir una "ráfaga" de datos al búfer sin preocuparse por sobrescribir los datos antiguos. El núcleo de E/S puede verificar el estado de FIFO con la señal `empty` de FIFO, eliminar datos con la señal `rd` de FIFO y procesar los datos a su propio ritmo.

El diagrama de bloques de la interfaz de lectura modificada se muestra en la Ilustración 16. El diseño agrega un circuito para eliminar datos de un FIFO después de la operación de lectura. El circuito es un decodificador idéntico al de la interfaz de escritura, excepto por la señal `write`, que es reemplazada por la señal `read`. La salida descodificada habilita la señal `rd` de FIFO designada para un reloj y elimina los datos recuperados previamente.

En una configuración más completa, se pueden agregar circuitos adicionales para recuperar el estado FIFO y ejercer un mejor control. Esto se demuestra en el circuito de envoltura del núcleo UART en la sección 5.4.3.2.

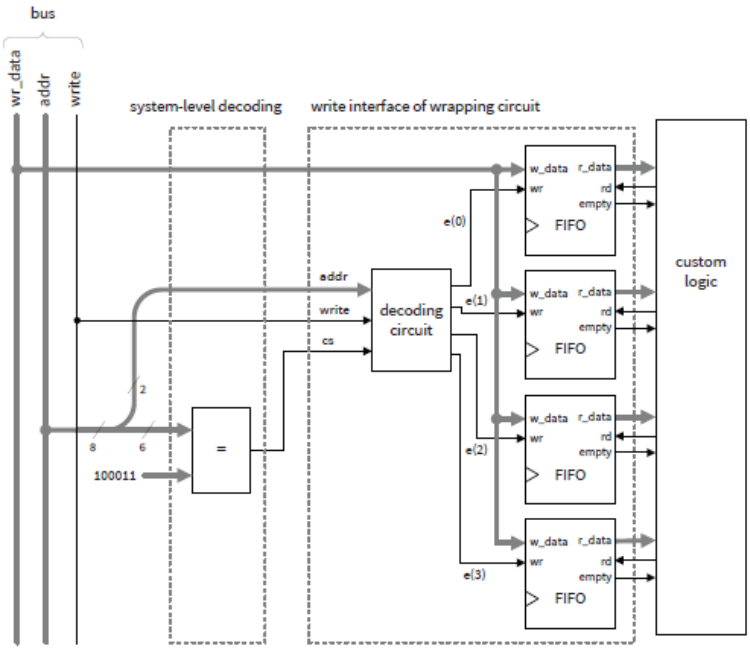


Ilustración 15: *Interfaz de escritura con buffers FIFO.*

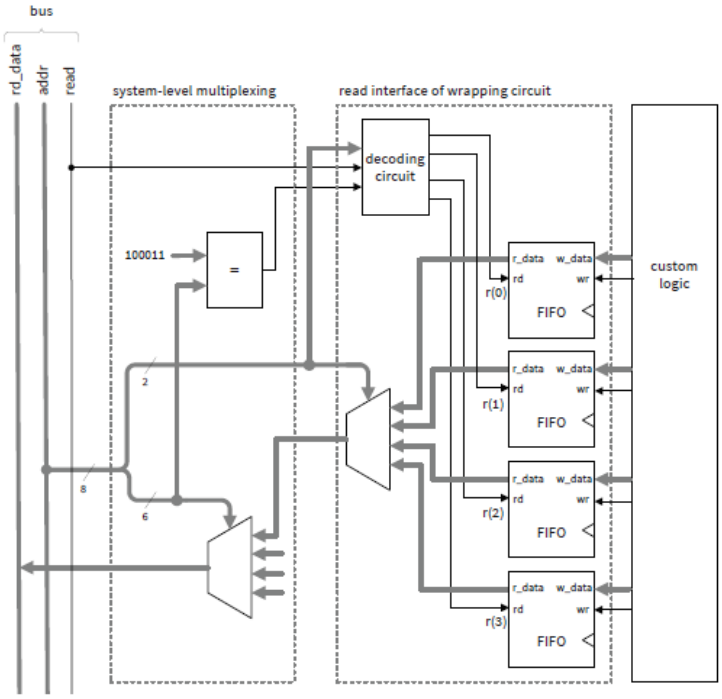


Ilustración 16: *Interfaz de lectura con buffers FIFO.*

5.3.2.5 Consideración de tiempo

Con un mejor conocimiento del circuito de interfaz con el bus, se puede estudiar con más detalle la restricción de sincronización del bus FPro. Se trata de una restricción de tiempo que se debe superar para lograr trabajar de forma segura y sin la aparición de errores con los datos y direcciones del bus. El diagrama de tiempo se muestra en la Ilustración 12.

A continuación se explica considerando la operación de escritura:

- En *t1*, el maestro emite un comando de escritura.
- En *t2*, todas las señales de bus son estables después de que transcurra el tiempo t_{CQ} (retardo desde el flanco de reloj, a las salidas Q) y se activaría el circuito de decodificación.
- En *t3*, la señal de habilitación decodificada alcanza el registro de E/S esclavo designado después del retardo t_{DEC}.
- En *t4*, el registro de E/S esclavo designado muestra y clasifica los datos de escritura.

Denominando al período de reloj como t_{CLK} y al tiempo que necesitan estar estables los datos del registro como t_{SETUP}. El circuito debe satisfacer esta imprescindible restricción de tiempo, en la que el flanco de reloj llega con posterioridad al tiempo que necesitan los datos para que se encuentren disponibles de forma estable:

$t_{CQ} + t_{DEC} + t_{SETUP} < t_{CLK}$

En otras palabras, t_{DEC} debe ser más pequeño que $t_{CLK} - (t_{CQ} + t_{SETUP})$.

El análisis para la operación de lectura es similar:

- En *t5*, el maestro emite un comando de lectura.
- En *t6*, todas las señales de bus y las salidas de registro de E/S son estables después de que t_{CQ} y el circuito de multiplexación se inician.
- En *t7*, los datos de lectura de origen designados se enrutan a las líneas de datos de lectura después de la demora de t_{MUX}.
- En *t8*, el maestro muestrea y recupera los datos del bus.

La restricción de temporización se convierte en

$t_{CQ} + t_{MUX} + t_{SETUP} < t_{CLK}$

Implica que t_{MUX} debe ser más pequeño que t_{CLK}- (t_{CQ} + t_{SETUP}).

Para nuestro caso, ambas restricciones pueden satisfacerse con un reloj de 100MHz. Dado que las operaciones de decodificación y multiplexación del bus FPro podrían ser complejas, llegado el caso se podría usar un reloj del sistema más lento, si fuera necesario.

5.3.3 BLOQUE DE E/S DEL SUBSISTEMA MMIO

El subsistema MMIO está compuesto por un controlador MMIO y una colección de núcleos, periféricos o bloques de E/S MMIO. Un bloque de E/S que cumpla con la interfaz de slot se puede conectar al controlador y el procesador puede acceder a él. El diagrama de bloques de un subsistema MMIO se muestra en la Ilustración 17.

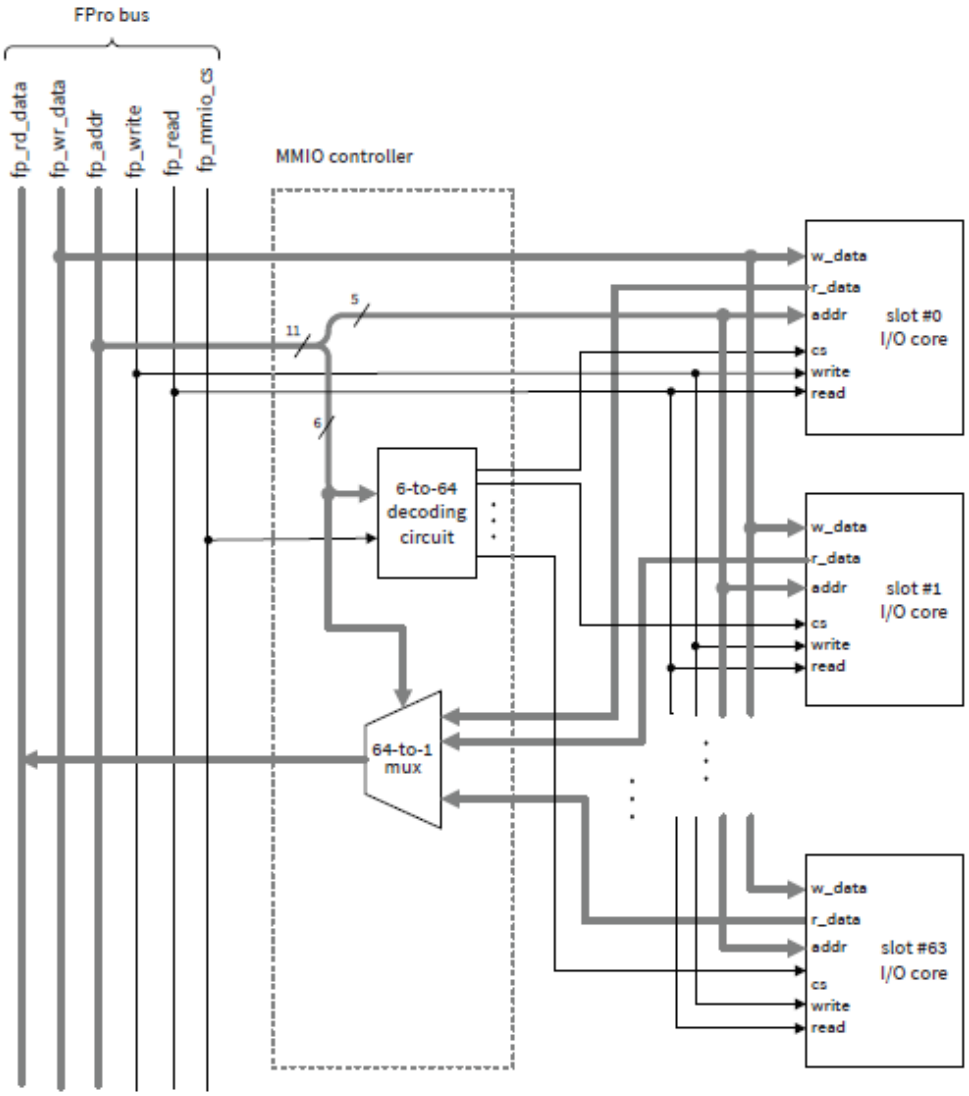


Ilustración 17: *Diagrama de bloques de un subsistema MMIO.*

5.3.3.1 Especificación de interfaz de ranura MMIO

Desde la perspectiva del procesador, un slot o ranura es un módulo de memoria de 32 palabras (2^5 palabras). La interfaz para el módulo se define de la siguiente manera:

- `addr` (del bus al núcleo). Es una señal de dirección de 5 bits que se utiliza para identificar el registro de E/S de destino de 32 bits dentro del bloque.
- `rd_data` (del módulo al bus). Es una señal de 32 bits que transporta los datos leídos.
- `wr_data` (del bus al módulo). Es una señal de 32 bits que lleva los datos de escritura.
- `read` (del bus al módulo). Es una señal de control de 1 bit activada con la operación de lectura.
- `write` (del bus al módulo). Es una señal de control de 1 bit para habilitar la escritura de registro.
- `cs` (del bus al módulo). Es una señal de habilitación de 1 bit ("*chip select*") para seleccionar y activar el módulo.

Estas características son similares a las del bus FPro, excepto que contiene una dirección de 5 bits más pequeña.

Recordemos que la anchura de las líneas de dirección y de datos ss fija. Los 32 registros representan el número máximo permitido en un bloque y no todos los registros tienen que utilizarse, algunos pueden dejarse sin usar. El ancho de datos de 32 bits se utiliza para facilitar el acceso a los datos del procesador. Si un elemento de datos tiene menos de 32 bits (por ejemplo, un byte), los bits no utilizados se ignorarán. Por otro lado, si un elemento de datos tiene más de 32 bits, este se debe dividir en varios registros y acceder a través de múltiples operaciones de lectura o escritura. El software debe empaquetar o desempaquetar los datos según sea necesario.

Debido a la diversidad de situaciones, se necesitaría para cada caso un circuito de decodificación y un circuito de multiplexación personalizados a nivel del sistema para cada diseño de SoC individual. Para simplificar y dar uniformidad, usamos una interfaz fija para el sistema FPro.

5.3.3.2 Construcción básica de un núcleo de E/S MMIO

La construcción de un núcleo de E/S MMIO constaría de los siguientes pasos:

1. Diseño el circuito digital personalizado.

2. Determinación del mapa de registros de E/S para la interfaz de memoria de la ranura.

3. Diseñar e implementar el circuito de envoltura.

4. Desarrollar el controlador de software.

Después de desarrollar un circuito digital personalizado, debemos determinar cómo interactúa el procesador con el núcleo de E/S, tales como escribir datos, leer datos, recuperar el estado y emitir comandos, y obtener un mapa de registro en consecuencia.

Como ya se ha comentado, un circuito de envoltura "envuelve" la lógica personalizada para crear una interfaz compatible con la especificación de la ranura. La "lógica envuelta" puede entonces insertarse en una ranura del controlador MMIO.

Sin embargo, la interfaz de un núcleo de E/S real no suele ser homogénea. Contiene señales de entrada y salida de diferentes anchos, características de acceso y restricciones de tiempo. Basándonos en la especificación del mapa de registro, podemos agregar los registros necesarios, los buffers FIFO, los circuitos de decodificación y los circuitos de multiplexación en el circuito de envoltura para que coincidan con la especificación de la ranura.

La declaración de la entidad de un núcleo de E/S MMIO debe incluir las señales de interfaz de ranura y las señales externas necesarias. Por ejemplo, la declaración de identidad del núcleo UART en VHDL es:

```
entity chu_uart is
   generic(
      FIFO_DEPTH_BIT : integer := 8 -- # FIFO addr bits
   );
   port(
      clk     : in  std_logic;
      reset   : in  std_logic;
      -- slot interface
      cs      : in  std_logic;
      write   : in  std_logic;
      read    : in  std_logic;
      addr    : in  std_logic_vector(4 downto 0);
```

```
      rd_data : out std_logic_vector(31 downto 0);
      wr_data : in  std_logic_vector(31 downto 0);
      -- external signals
      tx      : out std_logic;
      rx      : in  std_logic
   );
end chu_uart;
```

5.3.3.3 Núcleos GPO y GPI

Esta sección ilustra el diseño de los núcleos de GPO y GPI y la siguiente sección trata sobre el desarrollo de un núcleo de temporizador más sofisticado.

Núcleo de GPO. El núcleo de GPO constituye un puerto de salida de propósito general. Es simplemente un registro que mantiene los datos de salida. La composición de E/S de este módulo se muestra en la imagen a continuación:

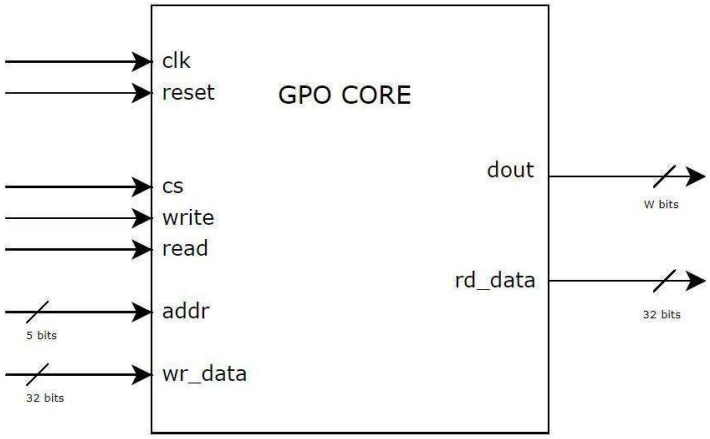


Ilustración 18: *Composición de E/S bloque GPO*

El código HDL se muestra en el Listado 3.1.

Listado 3.1. Bloque GPO

```
library ieee;
use ieee.std_logic_1164.all;
entity chu_gpo is
   generic(W : integer := 8);  -- width of output port
   port(
      clk     : in  std_logic;
      reset   : in  std_logic;
      -- slot interface
      cs      : in  std_logic;
      write   : in  std_logic;
      read    : in  std_logic;
      addr    : in  std_logic_vector(4 downto 0);
      rd_data : out std_logic_vector(31 downto 0);
      wr_data : in  std_logic_vector(31 downto 0);
      -- external signal
      dout    : out std_logic_vector(W - 1 downto 0)
   );
end chu_gpo;
```

```
architecture arch of chu_gpo is
   signal buf_reg : std_logic_vector(W - 1 downto 0);
   signal wr_en   : std_logic;
begin
   -- output buffer register
   process(clk, reset)
   begin
      if (reset = '1') then
         buf_reg <= (others => '0');
      elsif (clk'event and clk = '1') then
         if wr_en = '1' then
            buf_reg <= wr_data(W - 1 downto 0);
         end if;
      end if;
   end process;
   -- decoding logic
   wr_en   <= '1' when write = '1' and cs = '1' else '0';
   -- slot read interface
   rd_data <= (others => '0');          -- not used
   -- external output
   dout    <= buf_reg;
  end arch
```

El núcleo contiene un registro de datos de salida, `buf_reg`. La decodificación se realiza con la expresión `cs` = '1' y `write` = '1'. Los datos de escritura se almacenan en el registro cuando se afirma la condición. Como solo hay un registro de salida en el núcleo, la señal `addr` no se usa para la decodificación. Es posible o asignar un desplazamiento de dirección, digamos 0, para el registro de datos y descodificar esta dirección:

```
wr_en <= ‘1’ when write=’1’ and cs=’1’ and addr=”00000” else ‘0’;
```

Dado que la señal `rd_data`, no se usa, está conectada a 0`s. La señal no utilizada se optimizará durante la síntesis.

Bloque GPI: el bloque GPI constituye un puerto de entrada de propósito general. Contiene un registro de entrada que muestrea y almacena datos de entrada a cada flanco de reloj. La composición de E/S de este módulo se muestra en la imagen a continuación:

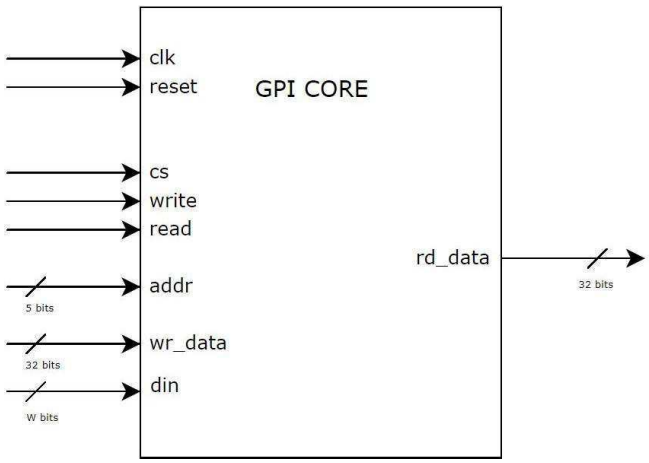


Ilustración 19: *Composición de E/S bloque GPI*

El código HDL se muestra en el Listado 3.2.

Listado 3.2. Bloque GPI

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity chu_gpi is
   generic(W : integer := 8);  -- width of input port
   port(
      clk     : in  std_logic;
      reset   : in  std_logic;
      -- slot interface
      cs      : in  std_logic;
      write   : in  std_logic;
      read    : in  std_logic;
      addr    : in  std_logic_vector(4 downto 0);
      rd_data : out std_logic_vector(31 downto 0);
      wr_data : in  std_logic_vector(31 downto 0);
      -- external signal
      din     : in  std_logic_vector(W-1 downto 0)
   );
end chu_gpi;

architecture arch of chu_gpi is
   signal rd_data_reg : std_logic_vector(W-1 downto 0);
begin
   -- input register
   process(clk, reset)
   begin
      if reset = '1' then
         rd_data_reg <= (others => '0');
      elsif (clk'event and clk = '1') then
         rd_data_reg <= din;
      end if;
   end process;
   -- slot read interface
   rd_data(W-1 downto 0) <= rd_data_reg;
   rd_data(31 downto W)  <= (others => '0');
  end arch
```

5.3.4 DESARROLLO DEL BLOQUE TEMPORIZADOR

Un núcleo temporizador es básicamente un contador con el que podemos determinar el tiempo transcurrido, si cuenta pulsos de una frecuencia patrón conocida, a través del valor de conteo alcanzado. Está compuesto por una serie de E/S que se muestran en la siguiente imagen:

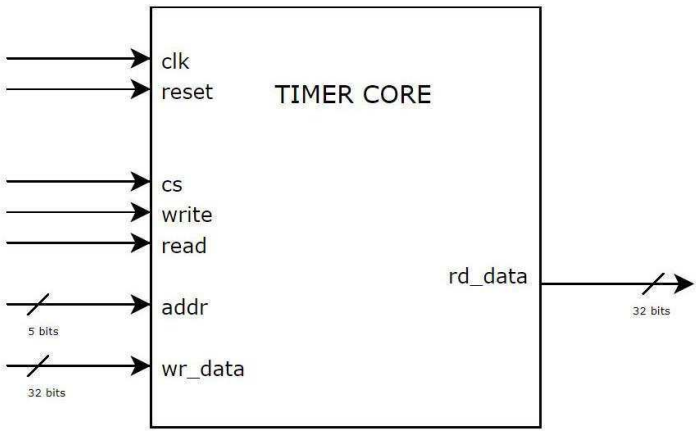


Ilustración 20: *Composición de E/S bloque timer*

5.3.4.1 Lógica personalizada

La lógica personalizada del núcleo del temporizador es un contador. El siguiente segmento de código es un contador de 48 bits que se puede borrar y pausar con las señales `clear` y `go`:

```
-- register
process(clk, reset)
begin
   if (reset = '1') then
      count_reg <= (others => '0');
   elsif (clk'event and clk = '1') then
      count_reg <= count_next;
   end if;
end process;
-- next-state logic
count_next <= (others=>'0') when clear = '1' else
              count_reg + 1 when go = '1' else
              count_reg;
```

El contador de 48 bits puede contar hasta 2^{48} ciclos de reloj. Con una frecuencia de reloj del sistema de 100MHz, puede contar hasta aproximadamente 32 días.

5.3.4.2 Mapa de registro

El procesador interactúa con el contador de la siguiente manera:

- Obtener (leer) el valor del contador de 48 bits.
- Configurar o restablecer (escribir) la señal de `go` para reanudar o pausar el conteo.
- Generar (escribir) un pulso para resetear el contador a valor 0.

Basándose en estas interacciones, se puede definir el mapa de registro del núcleo del temporizador. Hay dos registros de lectura. Sus desplazamientos y campos de dirección son:

- Offset 0 (palabra inferior del contador)
    - Bits 31 a 0: 32 LSBs del contador.
- Offset 1 (palabra superior del contador)
    - Bits 15 a 0: 16 MSBs del contador.

Hay un registro de escritura. Su dirección de desplazamiento y campos son:

- offset 2 (registro de control)
    - Bit 0: la señal de go del contador.
    - bit 1: la señal del contador.

5.3.4.3 Circuito de envoltura para la interfaz de ranura.

Sobre la base del mapa de registro y las señales de E/S del contador, podemos describir un circuito de envoltura que se sintetice con la especificación de la ranura y crear el bloque del temporizador. El código VHDL se muestra en el Listado 3.3.

Listado 3.3. Núcleo Temporizador

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity chu_timer is
   port(
      clk     : in  std_logic;
      reset   : in  std_logic;
      -- slot interface
      cs      : in  std_logic;
      write   : in  std_logic;
      read    : in  std_logic;
      addr    : in  std_logic_vector(4 downto 0);
      rd_data : out std_logic_vector(31 downto 0);
      wr_data : in  std_logic_vector(31 downto 0)
   );
end chu_timer;

architecture arch of chu_timer is
   signal count_reg  : unsigned(47 downto 0);
   signal count_next : unsigned(47 downto 0);
   signal ctrl_reg   : std_logic;
   signal wr_en      : std_logic;
   signal clear, go  : std_logic;
begin
   --*****************************************************************
   -- counter
   --*****************************************************************
   -- register
   process(clk, reset)
   begin
      if reset = '1' then
         count_reg <= (others => '0');
      elsif (clk'event and clk = '1') then
         count_reg <= count_next;
      end if;
   end process;
   -- next-state logic
   count_next <= (others => '0') when clear = '1' else
                 count_reg + 1   when go = '1' else
                 count_reg;

   --*****************************************************************
   -- wrapping circuit
   --*****************************************************************
  -- ctrl register
   process(clk, reset)
   begin
      if reset = '1' then
         ctrl_reg <= '0';
```

```
    elsif (clk'event and clk = '1') then
       if wr_en = '1' then
          ctrl_reg <= wr_data(0);
       end if;
    end if;
 end process;
 -- decoding logic
 wr_en <=
    '1' when write='1' and cs='1' and addr(1 downto 0)="10" else '0';
 clear <= '1' when wr_en='1' and wr_data(1)='1' else '0';
 go    <= ctrl_reg;
 -- slot read multiplexing
 rd_data <=
    std_logic_vector(count_reg(31 downto 0)) when addr(0)='0' else
    x"0000" & std_logic_vector(count_reg(47 downto 32));
end arch
```

El circuito de envoltura consta de un registro de control de un bit, un circuito de decodificación y un circuito de multiplexación de lectura. El registro de control mantiene el valor de la señal `go`. El circuito de decodificación decodifica los dos LSB de `addr` para generar una señal de habilitación de escritura, `wr_en`. Cuando se asegura, el bit 0 de `wr_data` se almacena en el registro de control. Tenga en cuenta que el bit 1 de `wr_data` no se mantiene. Se utiliza junto con `wr_en` para generar un flanco de un reloj para restablecer el contador a 0. El circuito de multiplexación usa el bit 0 de `addr` (para desplazamientos de 0 o 1) como señal de selección y dirige la palabra menos significativa o más significativa a `rd_data` Como el contador solo tiene 48 bits de ancho, 16 ceros se rellenan en los bits altos de la palabra más significativa.

5.3.5 CONTROLADOR MMIO

Como ya se indicó, el controlador MMIO puede incorporar hasta un máximo de 64 ranuras. Realiza la decodificación y multiplexación a nivel de subsistema y sirve como un circuito de interconexión entre el bus FPro y los núcleos de E/S. Un núcleo de E/S MMIO se puede conectar a una ranura y el procesador puede acceder a él.

El subsistema MMIO contiene hasta 64 (2^6) núcleos de E/S, cada uno con hasta 32 (2^5) registros de E/S. Desde el procesador, todo el subsistema se puede considerar como un módulo de E/S independiente con un espacio de dirección de 2^{11} bits, en el que los seis MSB son los bits de módulo utilizados para identificar un núcleo y los cinco LSB son bits de compensación utilizados para identificar un registro de E/S dentro del núcleo.

El controlador MMIO utiliza los seis bits de módulo para seleccionar y habilitar el núcleo de E/S designado y para realizar la decodificación y multiplexación a nivel de subsistema. El circuito de decodificación genera 64 señales de habilitación, cada una de las cuales se conecta a la señal de *Chip Select* (`cs`) de un bloque de E/S. El circuito de multiplexación es un multiplexor de 64 a 1 y encamina los datos de lectura desde el núcleo de E/S designado al bus FPro.

5.3.5.1 Archivo `chu_io_map.vhd`

Un paquete VHDL contiene definiciones constantes, definiciones de tipos de datos y subprogramas para ser compartidos por múltiples diseños. Un paquete comprende una declaración de paquete obligatoria, en la que se declaran las constantes, los tipos de datos y los subprogramas, y un cuerpo de paquete opcional, en el que se define la implementación del subprograma. La sintaxis de una declaración de paquete es

```
package package_name is
    constant_declaration;
    . . .
    Data_type_declaration;
    . . .
end package_name;
```

Después de construir un paquete, otros diseños pueden invocarlo con una declaración de uso, como en

```
use work.package_name.all;
```

Utilizamos un paquete VHDL, chu_io_map, para facilitar la construcción del subsistema MMIO y el subsistema de video. Los paquetes tienen dos propósitos. Primero, mantiene una lista de nombres de ranuras simbólicas, que nos ayudan a evitar desajustes de ranuras entre los desarrollos de hardware y software, como se discutió para prevenir desajustes de ranura entre los desarrollos de hardware y software. En segundo lugar, define tipos de datos bidimensionales para acomodar las declaraciones de puertos de salida del controlador MMIO y el controlador de video. Como no se define ningún subprograma, no se necesita ningún cuerpo del paquete. El código de la declaración del paquete se muestra en el Listado 3.4. Solo se incluye la parte relevante para el sistema de FPro.

Listado 3.4. Tipos de datos y declaraciones en el paquete `chu_io_map`

```
library ieee;
use ieee.std_logic_1164.all;

package chu_io_map is
   -- ****************************************************************
   -- 2D data types
   -- ****************************************************************
   type slot_2d_data_type is array (63 downto 0) of
        std_logic_vector(31 downto 0);
   type slot_2d_reg_type is array (63 downto 0) of
        std_logic_vector(4 downto 0);

   -- ****************************************************************
   -- Base address for the io_bridge
   -- ****************************************************************
   -- for xilinx MCS
   constant BRIDGE_BASE : std_logic_vector(31 downto 0) := X"c7000000";

   -- ****************************************************************
```

```
   -- slot definition for the "sampler" MMIO subsystem
   -- avoid changing the first four slots
   -- ************************************************************
   constant S0_SYS_TIMER : integer := 0;
   constant S1_UART1     : integer := 1;
   constant S2_LED       : integer := 2;
   constant S3_SW        : integer := 3;
   constant S4_USER      : integer := 4;
   constant S5_XADC      : integer := 5;
   constant S6_PWM       : integer := 6;
   constant S7_BTN       : integer := 7;
   constant S8_SSEG      : integer := 8;

   -- ************************************************************
   -- addional slot definition for the "dlx" MMIO subsystem
   -- ************************************************************
   constant S14_USER1 : integer := 14;
   constant S15_USER2 : integer := 15;
   constant S16_TIMER1 : integer := 16;
   constant S17_TIMER2 : integer := 17;
   constant S18_UART2 : integer := 18;
   constant S19_UART3 : integer := 19;
end chu_io_map;
```

5.3.5.2 Código HDL

El código HDL del controlador MMIO se muestra en el Listado 3.5. Hay que considerar que el paquete `chu_io_map` se invoca para que los tipos de datos bidimensionales se puedan usar en la declaración del puerto.

Listado 3.5. Controlador MMIO

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use work.chu_io_map.all;
entity chu_mmio_controller is
   port(
      -- FPro bus
      mmio_cs             : in  std_logic;
      mmio_wr             : in  std_logic;
      mmio_rd             : in  std_logic;
      mmio_addr           : in  std_logic_vector(20 downto 0);
      mmio_wr_data        : in  std_logic_vector(31 downto 0);
      mmio_rd_data        : out std_logic_vector(31 downto 0);
      -- slot interface
      slot_cs_array       : out std_logic_vector(63 downto 0);
      slot_mem_rd_array   : out std_logic_vector(63 downto 0);
      slot_mem_wr_array   : out std_logic_vector(63 downto 0);
      slot_reg_addr_array : out slot_2d_reg_type;
      slot_rd_data_array  : in  slot_2d_data_type;
      slot_wr_data_array  : out slot_2d_data_type
   );
end chu_mmio_controller;

architecture arch of chu_mmio_controller is
   -- 11 LSBs of address used; 2^6 slots, each with 2^5 registers
   alias slot_addr : std_logic_vector(5 downto 0) is
         mmio_addr(10 downto 5);
   alias reg_addr  : std_logic_vector(4 downto 0) is
         mmio_addr(4 downto 0);
begin
   -- address decoding
      process(slot_addr, mmio_cs)
   begin
      slot_cs_array <= (others => '0');
      if mmio_cs = '1' then
         slot_cs_array(to_integer(unsigned(slot_addr))) <= '1';
      end if;
   end process;
```

```
   -- broadcast to all slots
   slot_mem_rd_array   <= (others => mmio_rd);
   slot_mem_wr_array   <= (others => mmio_wr);
   slot_wr_data_array  <= (others => mmio_wr_data);
   slot_reg_addr_array <= (others => reg_addr);
   -- mux for read data
   mmio_rd_data <= slot_rd_data_array(to_integer(unsigned(slot_addr)));
end arch;
```

Las entradas son las señales del bus FPro y las salidas son 64 interfaces de ranura que se conectan a 64 núcleos de E/S. Hay que tener en cuenta que el bus FPro tiene una línea de dirección de 21 bits (conectada a `mmio_addr`) pero el controlador MMIO solo utiliza 11 bits. Los 11 bits se dividen en `slot_addr` (seis bits de módulo) y `reg_addr` de 5 bits (los cinco bits de desplazamiento). La construcción de alias VHDL se utiliza para aclarar el código. La declaración simplemente extrae una parte de una señal y le da un nuevo nombre.

Las partes clave del controlador son el decodificador de 6 a 64 y el multiplexor de 64 a 1. Se codifican utilizando indexación dinámica. El segmento de código del circuito de multiplexación es

`mmio_rd_data` **<=** `slot_rd_data_array` (... `slot_addr` ...);

El segmento de código del circuito de decodificación es

```
slot_cs_array <= (others => '0');
if mmio_cs = '1' then
  slot_cs_array(to_integer(unsigned(slot_addr))) <= ‘1’;
end if;
```

Todos los elementos de `slot_cs_array` se asignan a '0'. Si el subsistema MMIO está habilitado `(mmio_cs = '1')`, la señal `cs` de la ranura designada se sobrescribirá y se confirmará. Otras señales del bus FPro, que incluyen `mmio_rd`, `mmio_wr`, `mmio_wr_data` y los cinco LSB de `mmio_addr` `(reg_addr)`, se vuelven a transmitir a todas las ranuras.

5.3.5.3 Subsistema MMIO

El subsistema MMIO de vanilla está compuesto por un controlador MMIO y cuatro núcleos de E/S, que son el núcleo del temporizador, el núcleo UART, el núcleo GPO y el núcleo GPI. Luego se utilizan enlaces para crear el sistema de vanilla FPro. El diagrama de bloques del sistema FPro se muestra en la Ilustración 21 y el subsistema MMIO de está en la parte inferior. Los nombres de los módulos en letra cursiva corresponden a los de los códigos HDL.

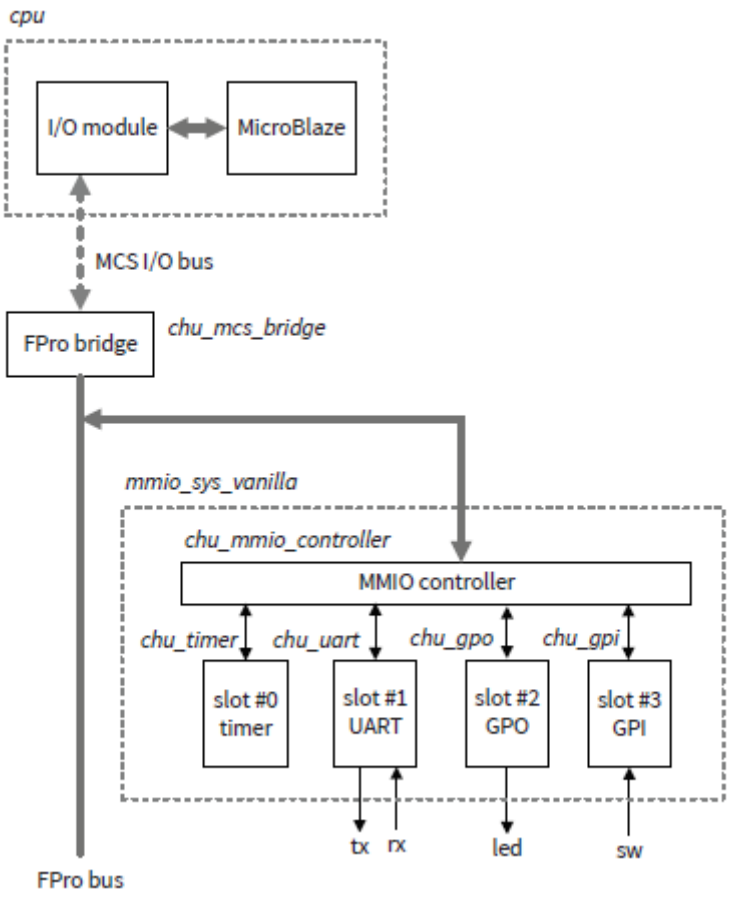


Ilustración 21: *Sistema Vanilla FPro.*

El código HDL se muestra en el Listado 3.6. El diseño detallado y la codificación del bloque UART se tratan en la Sección 5.4.

Listado 3.6. Subsistema MMIO vanilla

```
library ieee;
use ieee.std_logic_1164.all;
use work.chu_io_map.all;
entity mmio_sys_vanilla is
   generic(
     N_LED: integer;
     N_SW: integer
   );
   port(
      -- FPro bus
      clk          : in  std_logic;
      reset        : in  std_logic;
      mmio_cs      : in  std_logic;
      mmio_wr      : in  std_logic;
      mmio_rd      : in  std_logic;
      mmio_addr    : in  std_logic_vector(20 downto 0); -- only 11 LSBs used
      mmio_wr_data : in  std_logic_vector(31 downto 0);
      mmio_rd_data : out std_logic_vector(31 downto 0);
      -- switches and LEDs
      sw           : in  std_logic_vector(N_SW-1 downto 0);
      led          : out std_logic_vector(N_LED-1 downto 0);
      -- uart
      rx           : in  std_logic;
      tx           : out std_logic
   );
end mmio_sys_vanilla;

architecture arch of mmio_sys_vanilla is
   signal cs_array       : std_logic_vector(63 downto 0);
   signal reg_addr_array : slot_2d_reg_type;
   signal mem_rd_array   : std_logic_vector(63 downto 0);
   signal mem_wr_array   : std_logic_vector(63 downto 0);
   signal rd_data_array  : slot_2d_data_type;
   signal wr_data_array  : slot_2d_data_type;
begin
   --*****************************************************************
   --  MMIO controller instantiation
   --*****************************************************************
   ctrl_unit : entity work.chu_mmio_controller
      port map(
         -- FPro bus interface
         mmio_cs             => mmio_cs,
         mmio_wr             => mmio_wr,
         mmio_rd             => mmio_rd,
         mmio_addr           => mmio_addr,
         mmio_wr_data        => mmio_wr_data,
         mmio_rd_data        => mmio_rd_data,
         -- 64 slot interface
         slot_cs_array       => cs_array,
         slot_reg_addr_array => reg_addr_array,
         slot_mem_rd_array   => mem_rd_array,
         slot_mem_wr_array   => mem_wr_array,
         slot_rd_data_array  => rd_data_array,
         slot_wr_data_array  => wr_data_array
      );

   --*****************************************************************
   -- IO slots instantiations
   --*****************************************************************
   -- slot 0: system timer
   timer_slot0 : entity work.chu_timer
      port map(
         clk           => clk,
         reset         => reset,
         cs            => cs_array(S0_SYS_TIMER),
         read          => mem_rd_array(S0_SYS_TIMER),
         write         => mem_wr_array(S0_SYS_TIMER),
         addr          => reg_addr_array(S0_SYS_TIMER),
         rd_data       => rd_data_array(S0_SYS_TIMER),
         wr_data       => wr_data_array(S0_SYS_TIMER)
      );
```

```
-- slot 1: uart1
uart1_slot1 : entity work.chu_uart
   generic map(FIFO_DEPTH_BIT => 6)
   port map(
      clk     => clk,
      reset   => reset,
      cs      => cs_array(S1_UART1),
      read    => mem_rd_array(S1_UART1),
      write   => mem_wr_array(S1_UART1),
      addr    => reg_addr_array(S1_UART1),
      rd_data => rd_data_array(S1_UART1),
      wr_data => wr_data_array(S1_UART1),
      -- external signals
      tx      => tx,
      rx      => rx
   );
-- slot 2: GPO for LEDs
gpo_slot2 : entity work.chu_gpo
   generic map(W => N_LED)
   port map(
      clk     => clk,
      reset   => reset,
      cs      => cs_array(S2_LED),
      read    => mem_rd_array(S2_LED),
      write   => mem_wr_array(S2_LED),
      addr    => reg_addr_array(S2_LED),
      rd_data => rd_data_array(S2_LED),
      wr_data => wr_data_array(S2_LED),
      -- external signal
      dout    => led
   );
-- slot 3: input port for switches
gpi_slot3 : entity work.chu_gpi
   generic map(W => N_SW)
   port map(
      clk     => clk,
      reset   => reset,
      cs      => cs_array(S3_SW),
      read    => mem_rd_array(S3_SW),
      write   => mem_wr_array(S3_SW),
      addr    => reg_addr_array(S3_SW),
      rd_data => rd_data_array(S3_SW),
      wr_data => wr_data_array(S3_SW),
      -- external signal
      din     => sw
   );
-- assign 0's to all unused slot rd_data signals
gen_unused_slot : for i in 4 to 63 generate
   rd_data_array(i) <= (others => '0');
   end generate gen_unused_slot;
```

El código en primer lugar, crea una instancia del controlador MMIO y posteriormente crea las instancias de los cuatro núcleos o bloques. Un bloque se "inserta" en un slot del controlador asignando su interfaz a un elemento específico de la matriz de 64 ranuras del controlador. Una constante simbólica definida en el archivo `chu_io_map.vhd`, como `S1_UART1`, se usa para el número de ranura. Para las 60 ranuras no utilizadas, una declaración de generación vincula sus puertos de lectura a 0´s. Estos serán optimizados en la síntesis.

5.3.6 MCS E/S BUS Y PUENTE

5.3.6.1 Descripción general de Xilinx MicroBlaze MCS

MicroBlaze es un procesador “core soft” de 32 bits proporcionado por Xilinx. El término "core soft" en este contexto significa que el procesador se construye a partir de las celdas lógicas y recursos programables del dispositivo FPGA. MicroBlaze es altamente configurable y se pueden incluir una variedad de características opcionales. Utiliza la interfaz AXI para comunicarse con otros núcleos IP. Se puede integrar una gran colección de núcleos IP prediseñados, incluidos controladores de memoria, periféricos de E/S generales y aceleradores de hardware especializados, para formar un diseño de SoC.

MicroBlaze MCS (sistema microcontrolador) es un sistema basado en MicroBlaze que contiene un procesador MicroBlaze pre-configurado, memoria local y un módulo de E/S, que contiene un conjunto de periféricos de E/S estándar y un "puerto de bus". MCS está destinado a ser utilizado como un microcontrolador de 32 bits.

MicroBlaze MCS puede utilizarse a partir de la creación de una instancia con la utilidad de catálogo IP de Vivado. Para nuestros propósitos, se debe configurar de la siguiente manera:

- Establecer el tamaño de la memoria a 128 KB. Esto ayuda a ejecutar programas más grandes.
- Habilitar el puerto de bus de E/S del módulo de E/S. El puerto del bus se utilizará para puentear el bus FPro.
- Deseleccionar todos los demás periféricos de E/S. Los módulos de E/S se construirán desde cero en el subsistema MMIO.

5.3.6.2 Bus de E//S de MicroBlaze MCS

A diferencia del MicroBlaze, que contiene todas las funciones, MCS no admite la interfaz AXI. Utiliza un puerto de bus de E/S simple, que es el puerto del módulo de E/S, para comunicarse con componentes externos. Se asigna un espacio de direcciones de bytes de 30 bits, de 0xc0000000 a 0xffffffff, para este propósito.

El bus de E/S MCS es un bus síncrono. Además del reloj y reset del sistema, contiene las siguientes señales:

- `IO_Adress` (maestro a esclavo). Es una señal de 32 bits. Hay que tener en cuenta que el espacio de memoria del módulo de E/S MCS es "byte direccionable", lo que significa que la ubicación especificada por `IO_Address` es un byte.

- `IO_Read_Data` (esclavo a maestro). Es una señal de 32 bits que transporta los datos leídos.
- `IO_Write_Data` (maestro a esclavo). Es una señal de 32 bits que lleva los datos de escritura.
- `IO_Addr_Strobe` (maestro a esclavo). Es una señal de control de 1 bit para indicar una operación de lectura.
- `IO_Read_Strobe` (maestro a esclavo). Es una señal de control de 1 bit asociada con una operación de lectura.
- `IO_Write_Strobe` (maestro a esclavo) Es una señal de control de 1 bit para iniciar una operación de escritura.
- `IO_Byte_Enable` (maestro a esclavo). Es una señal de control de 4 bits para indicar qué bytes de `IO_Write_Data` se utilizan en la operación de escritura.
- `IO_Ready` (esclavo a maestro). Es una señal de estado de 1 bit del esclavo para indicar si la transacción designada se ha completado.

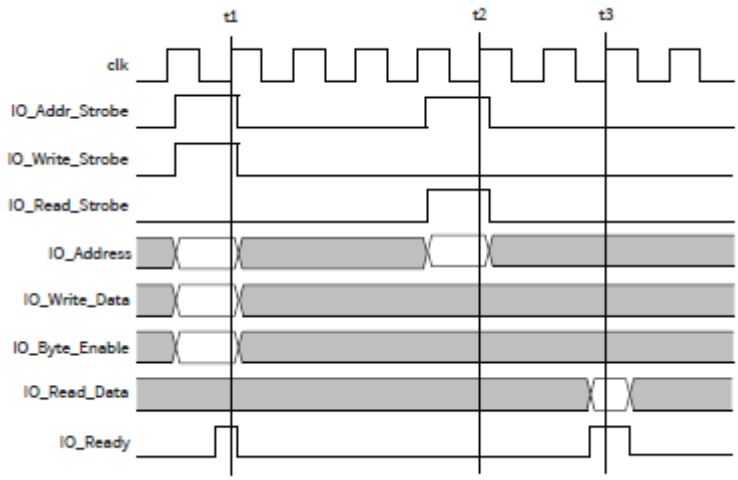


Ilustración 22: *Diagrama de tiempo representativo del bus de E/S MCS.*

En la parte izquierda de la Ilustración 22, se muestra el diagrama de sincronización de una operación típica de. El procesador coloca la dirección y escribe los datos en las líneas `IO_Address` y `IO_Write_Data` y asegura las señales `IO_Adrr_Strobe` y `IO_Write_Strobe` para iniciar la operación. El periférico de E/S designado procesa los datos. Una vez completado, informa al procesador poniendo en alto la señal `IO_Ready`. La `IO_Byte_Enable` se puede usar si solo se escribe una parte de la palabra.

El diagrama de sincronización de una operación de lectura típica se muestra en la parte derecha de la Figura 10.9. El procesador coloca la dirección en las líneas `IO_Address` y asegura las señales `IO_Adrr_Strobe` e `IO_Read_Strobe` para iniciar la operación. El periférico de E/S designado recupera sus datos internos. Una vez completado, coloca los datos de lectura en las líneas `IO_Read_Data` y pone en alto la señal `IO_Ready`.

El bus de E/S MCS utiliza la señal `IO_Ready` para implementar un protocolo “*handshake*”. Un esclavo puede mantener `IO_Ready` bajo cuando los datos no están listos y, por lo tanto, el bus puede incorporar dispositivos de E/S lentos en el sistema. Sin embargo, existe un problema potencial con el protocolo “*handshake*” en un entorno de creación de prototipos. Si un núcleo de E/S se comporta mal y no hace valer correctamente una señal de listo, el procesador seguirá esperando. Esto significa que un núcleo de E/S mal diseñado congelará todo el sistema.

5.3.6.3 Puente de MCS a FPro

El bloque de puente MCS a FPro "traduce" las transacciones de escritura y lectura del bus de E/S MCS a las operaciones correspondientes en el bus FPro. Como el protocolo del bus FPro es más simple que el protocolo del bus de E/S MCS, el diseño del puente es bastante sencillo. Realiza principalmente la conversión de direcciones y la conversión de líneas de control. El código HDL correspondiente, se muestra en el Listado 3.7.

Listado 3.7. Puente MCS a FPro

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use work.chu_io_map.all;
entity chu_mcs_bridge is
   generic(BRG_BASE : std_logic_vector(31 downto 0) := x"C0000000");
   port(
      -- uBlaze MCS I/O bus
      io_addr_strobe  : in  std_logic;  -- not used
      io_read_strobe  : in  std_logic;
      io_write_strobe : in  std_logic;
      io_byte_enable  : in  std_logic_vector(3 downto 0);
      io_address      : in  std_logic_vector(31 downto 0);
      io_write_data   : in  std_logic_vector(31 downto 0);
      io_read_data    : out std_logic_vector(31 downto 0);
      io_ready        : out std_logic;
      -- FPro bus
      fp_video_cs     : out std_logic;
      fp_mmio_cs      : out std_logic;
      fp_wr           : out std_logic;
      fp_rd           : out std_logic;
      fp_addr         : out std_logic_vector(20 downto 0);
      fp_wr_data      : out std_logic_vector(31 downto 0);
      fp_rd_data      : in  std_logic_vector(31 downto 0)
   );
end chu_mcs_bridge;

architecture arch of chu_mcs_bridge is
   signal mcs_bridge_en : std_logic;
   signal word_addr     : std_logic_vector(29 downto 0);
```

```
begin
   -- address translation and decoding
   -- 2 LSBs are "00" due to word alignment
   word_addr     <= io_address(31 downto 2);
   mcs_bridge_en <=
      '1' when io_address(31 downto 24)=BRG_BASE(31 downto 24) else '0';
   fp_video_cs   <=
      '1' when mcs_bridge_en='1' and io_address(23)='1' else '0';
   fp_mmio_cs    <=
      '1' when mcs_bridge_en='1' and io_address(23)='0' else '0';
   fp_addr       <= word_addr(20 downto 0);
   -- control line conversion
   fp_wr         <= io_write_strobe;
   fp_rd         <= io_read_strobe;
   io_ready      <= '1'; -- not used; transaction done in 1 clock
   -- data line conversion
   fp_wr_data    <= io_write_data;
   io_read_data  <= fp_rd_data;
end arch;
```

El circuito de traducción de direcciones decodifica la dirección de byte de 32 bits del bus de E/S MCS y la convierte en la dirección de palabra de 21 bits del bus FPro y las señales de selección de chip. Desde la perspectiva del procesador MCS, el sistema de E/S FPro es un módulo de E/S único con un espacio direccionable de 24 bits (espacio direccionable de 22 bits) en una dirección base de 0xc0000000. El circuito de traducción de direcciones se decodifica la dirección de 32 bits de la siguiente manera:

- bits 31 a 24: se utilizan para decodificar la dirección base del módulo de E/S FPro.

- bit 23: se utiliza para distinguir los dos subsistemas y generar las señales de selección de chip.

- bits 22 a 2: se utilizan para identificar un registro de E/S o una ubicación de memoria en el MMIO y subsistemas de video. Forman la dirección FPro de 21 bits.

- bits 1 a 0: no se utilizan porque el sistema FPro usa "dirección de palabra".

La dirección base se declara como una constante genérica, `BRG_BASE`, que es 0xc0000000 y se define en el paquete `chu_io_map`. Los ocho MSB de la dirección, que pueden considerarse como los bits del módulo, se utilizan para la decodificación a nivel del sistema. El `mcs_bridge_en` se activa si los bits de módulo del bus de E/S coinciden con los de la dirección base. Luego se usa para habilitar la señal de selección de chip del subsistema MMIO o subsistema de video (`fp_mmio_cs` o `fp_video_cs`).

El circuito de conversión de la línea de control pasa las señales de lectura y escritura directamente desde el bus de E/S MCS al bus FPro activando `IO_Ready` de forma continua (lo pone a 1). Desde la perspectiva del maestro, después de iniciar una operación de lectura o escritura, `IO_Ready` se activa

inmediatamente en el siguiente flanco ascendente del reloj. Implica que la operación se completa en un ciclo de reloj sin necesitar ningún intercambio más del bus de E/S MCS al bus FPro. La señal `IO_Addr_Strobe` no se verifica y la señal `IO_Byte_Enable` tampoco se usa, ya que se asume que la transacción de E/S siempre se realiza en una base de palabra, como se define en el `chu_io_wr.h` en la Sección 5.2.4.3.

Para cumplir con los requisitos de tiempo discutidos en la Sección 5.3.2.5, las señales de salida del puente deben almacenarse en búfer con registros de salida. Sin embargo, la documentación del módulo de E/S MCS indica que las señales del bus de E/S ya están en búfer y, por lo tanto, se omite el búfer en este puente en particular.

5.3.7 CONSTRUCCION SISTEMA VANILLA FPRO

El sistema vanilla FPro es una configuración funcional básica. No incluye el subsistema de video y su subsistema MMIO contiene solo los cuatro núcleos de E/S más esenciales. Es la base para sistemas más sofisticados. El diagrama de bloques se muestra en la Figura 3.8.

El código HDL se puede derivar siguiendo la jerarquía del diagrama de bloques. El subsistema MMIO de vanilla contiene el controlador MMIO y cuatro núcleos de E/S y se describe en la Sección 5.3.5.3. El sistema FPro de nivel superior consta de un procesador MCS, el puente de MCS a FPro y el subsistema MMIO de vanilla. El código HDL se muestra en el Listado 3.8.

Listado 3.8. Sistema FPro vanilla

```
library ieee;
use ieee.std_logic_1164.all;
use work.chu_io_map.all;
entity mcs_top_vanilla is
   generic(BRIDGE_BASE : std_logic_vector(31 downto 0) := x"C0000000");
   port(
      clk     : in  std_logic;
      reset_n : in  std_logic;
      -- switches and LEDs
      sw      : in  std_logic_vector(15 downto 0);
      led     : out std_logic_vector(15 downto 0);
      -- uart
      rx      : in  std_logic;
      tx      : out std_logic

   );
end mcs_top_vanilla;

architecture arch of mcs_top_vanilla is
   component cpu
      port(
         clk             : in  std_logic;
         reset           : in  std_logic;
         io_addr_strobe  : out std_logic;
         io_read_strobe  : out std_logic;
         io_write_strobe : out std_logic;
         io_address      : out std_logic_vector(31 downto 0);
         io_byte_enable  : out std_logic_vector(3 downto 0);
         io_write_data   : out std_logic_vector(31 downto 0);
```

```
        io_read_data    : in  std_logic_vector(31 downto 0);
        io_ready        : in  std_logic
      );
   end component;
   signal clk_100M        : std_logic;
   signal reset_sys       : std_logic;
   -- MCS IO bus
   signal io_addr_strobe  : std_logic;
   signal io_read_strobe  : std_logic;
   signal io_write_strobe : std_logic;
   signal io_byte_enable  : std_logic_vector(3 downto 0);
   signal io_address      : std_logic_vector(31 downto 0);
   signal io_write_data   : std_logic_vector(31 downto 0);
   signal io_read_data    : std_logic_vector(31 downto 0);
   signal io_ready        : std_logic;
   -- fpro bus
   signal fp_mmio_cs         : std_logic;
   signal fp_wr         : std_logic;
   signal fp_rd         : std_logic;
   signal fp_addr       : std_logic_vector(20 downto 0);
   signal fp_wr_data    : std_logic_vector(31 downto 0);
   signal fp_rd_data    : std_logic_vector(31 downto 0);
begin
   -- clock and reset
   clk_100M  <= clk;   -- 100 MHz external clock
   reset_sys <= not reset_n;
   -- instantiate microBlaze MCS
   mcs_0 : cpu
      port map(
         clk             => clk_100M,
         reset           => reset_sys,
         io_addr_strobe  => io_addr_strobe,
         io_read_strobe  => io_read_strobe,
         io_write_strobe => io_write_strobe,
         io_byte_enable  => io_byte_enable,
         io_address      => io_address,
         io_write_data   => io_write_data,
         io_read_data    => io_read_data,
         io_ready        => io_ready
      );
   -- instantiate MCS IO bus to FPro bus bridge
   bridge_unit : entity work.chu_mcs_bridge
      generic map(BRG_BASE => BRIDGE_BASE)
      port map(
         io_addr_strobe  => io_addr_strobe,
         io_read_strobe  => io_read_strobe,
         io_write_strobe => io_write_strobe,
         io_byte_enable  => io_byte_enable,
         io_address      => io_address,
         io_write_data   => io_write_data,
         io_read_data    => io_read_data,
         io_ready        => io_ready,
         fp_video_cs     => open,
         fp_mmio_cs      => fp_mmio_cs,
         fp_wr           => fp_wr,
         fp_rd           => fp_rd,
         fp_addr         => fp_addr,
         fp_wr_data      => fp_wr_data,
         fp_rd_data      => fp_rd_data
      );
   -- instantiate vanilla MMIO subsystem
   mmio_sys_unit : entity work.mmio_sys_vanilla
      generic map(
         N_LED=>16,
         N_SW=>16
      )
      port map(
         clk          => clk_100M,
         reset        => reset_sys,
         mmio_cs      => fp_mmio_cs,
         mmio_wr      => fp_wr,
         mmio_rd      => fp_rd,
         mmio_addr    => fp_addr,
         mmio_wr_data => fp_wr_data,
```

```
        mmio_rd_data => fp_rd_data,
        sw           => sw,
        led          => led,
        rx           => rx,
        tx           => tx
    );
end arch;
```

El código sigue el esquema anteriormente descrito y crea una instancia de los tres componentes.

5.4 BLOQUE UART

La comunicación en serie utiliza una sola línea de datos para intercambiar información entre dos sistemas. El sistema de transmisión convierte los datos paralelos a un flujo de transmisión en serie y el sistema de recepción vuelve a ensamblar los datos en serie a su formato paralelo original. Un UART (receptor y transmisor asíncrono universal) es el circuito más utilizado y la composición de E/S de un módulo se muestra en la Ilustración 23.

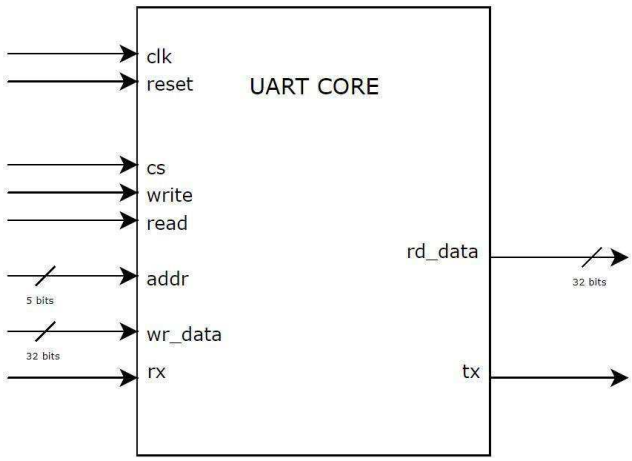


Ilustración 23: *Composición de E/S bloque UART*

5.4.1 INTRODUCCIÓN

5.4.1.1 Descripción general de la comunicación en serie

Los sistemas necesitan habitualmente comunicarse con otros que no residen en el mismo dispositivo. Para reducir la cantidad de pines de E/S y cableado externos, los dos sistemas pueden transferir datos a través de una sola línea serie, un bit a la vez. El sistema de transmisión realiza la conversión de paralelo a serie y luego envía los datos en serie a través de una sola línea. El sistema receptor realiza la conversión de serie a paralelo y restaura los datos paralelos originales.

La comunicación en serie se puede utilizar para la transferencia de datos de alta y baja velocidad. En una interfaz de alta velocidad, como USB y Gigabit Ethernet, la velocidad de datos puede alcanzar varios cientos de miles de millones de bits por segundo o más. Los circuitos de transmisión y recepción se conocen comúnmente como bloques SerDes (para serializador-deserializador).

Su protocolo y especificación es muy complejo y su diseño está fuera del alcance de este proyecto.

En una interfaz de baja velocidad, la velocidad de datos varía de varios miles a varios cientos de miles de bits por segundo. Es adecuada para un gran número de periféricos de E/S generales y para las tareas de adquisición y control de datos. Estos esquemas pueden realizarse mediante los elementos lógicos programables disponibles de forma genérica en los dispositivos FPGA.

5.4.1.2 Descripción general de UART

Un controlador UART básico incluye un transmisor y un receptor. El transmisor es un registro de desplazamiento especial que carga datos en paralelo y luego los desplaza bit a bit a una velocidad específica. El receptor, por otro lado, cambia los datos bit a bit y luego vuelve a montar los datos. La línea serie es 1 cuando está inactiva. La transmisión comienza con un bit de inicio, que es 0, seguido de bits de datos y un bit de paridad opcional, y finaliza con bits de parada, que es un 1. El número de bits de datos puede ser 6, 7 u 8. El bit de paridad (opcional) se utiliza para la detección de errores. Para una paridad impar, se establece en 0 cuando los bits de datos tienen un número impar de 1. Para una paridad par, se establece en 0 cuando los bits de datos tienen un número par de 1. El número de bits de parada puede ser 1, 1.5 o 2. La transmisión con 8 bits de datos, sin paridad y 1 bit de parada se muestra en la Ilustración 24. Hay que tener en cuenta que el LSB de la palabra de datos se transmite en primer lugar.

Ilustración 24: *Transmisión de un byte.*

No se transmite información de reloj a través de la línea serie. Antes de que comience la transmisión, el transmisor y el receptor deben acordar un conjunto de parámetros por adelantado, que incluyen la velocidad en baudios (número de bits por segundo), el número de bits de datos y los bits de parada, y el uso del bit de paridad. Con los parámetros predeterminados, el receptor utiliza un esquema de sobremuestreo para recuperar los bits de datos. Las velocidades en baudios de uso común son 9600, 19200 y 115200 baudios.

5.4.1.3 Procedimiento de sobremuestreo

La frecuencia de muestreo más utilizada es de 16 veces la velocidad en baudios, lo que significa que cada bit serie se muestrea 16 veces. Para una comunicación con N bits de datos y M bits de parada, el esquema de sobremuestreo funciona de la siguiente manera:

1. Esperar hasta que la señal entrante se convierta en 0, el comienzo del bit de inicio, y luego iniciar el contador de tick de muestreo.

2. Cuando el contador llega a 7, la señal entrante alcanza el punto medio del bit de inicio. Borrar el contador a 0 y reiniciar.

3. Cuando el contador llega a 15, la señal entrante avanza por un bit y llega a la mitad del primer bit de datos. Recuperar su valor, convertirlo en un registro y reiniciar el contador.

4. Repetir el paso 3, N-1 más veces para recuperar los bits de datos restantes.

5. Si se usa el bit de paridad opcional, repetir el paso 3 una vez para obtener el bit de paridad.

6. Repetir el paso 3, M veces más para obtener los bits de parada.

El esquema de sobremuestreo básicamente realiza la función de una señal de reloj. En lugar de utilizar el flanco ascendente para indicar cuándo la señal de entrada es válida, utiliza tics de muestreo para estimar el punto medio de cada bit. La estimación puede ser errónea en un máximo de $\frac{1}{16}$. A causa del sobremuestreo, la velocidad en baudios solo puede ser una pequeña fracción de la velocidad de reloj del sistema, y por lo tanto este esquema no es apropiado para una alta velocidad de datos.

5.4.2 CONSTRUCCION DE UART

5.4.2.1 Diseño conceptual

El diagrama de nivel superior de un sistema UART se muestra en la Ilustración 25. Se compone de cinco componentes principales. El generador de velocidad de transmisión genera una señal de sobremuestreo. El receptor realiza la conversión de serie a paralelo y el transmisor realiza la conversión de paralelo a serie. Se utilizan dos búferes FIFO entre el receptor y transmisor UART y el procesador como elementos “amortiguadores”.

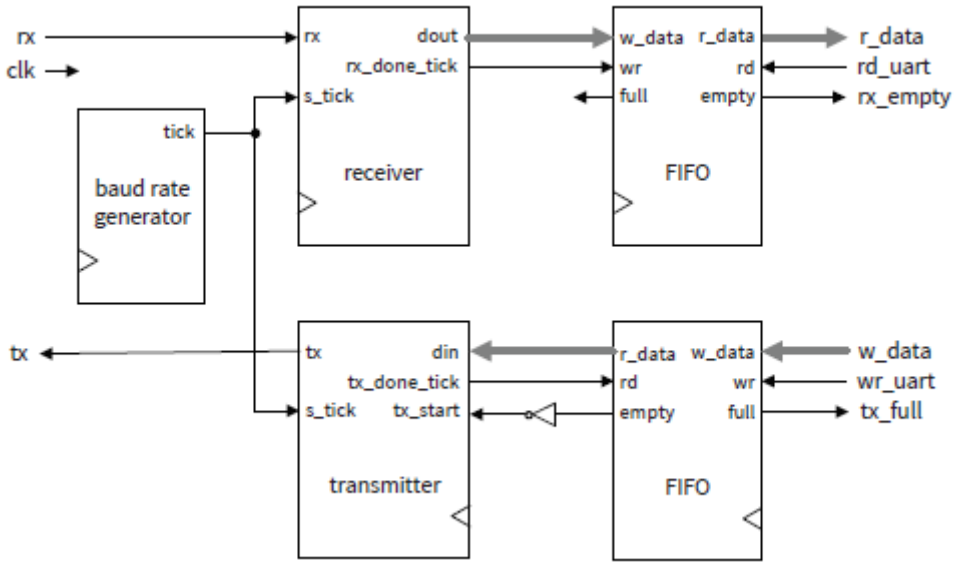


Ilustración 25: *Diagrama de bloques de un UART completo.*

Los buffers FIFO son necesarios dado que la tasa de procesamiento de datos de un UART es mucho menor que la tasa del procesador, aunque este sea de tipo “core soft” en una FPGA. Un búfer permite al procesador procesar una ráfaga de datos.

El diseño está personalizado para un UART sin un bit de paridad.

5.4.2.2 Generador de velocidad de transmisión

El generador de velocidad de transmisión (en baudios) genera una señal de muestreo cuya frecuencia es exactamente 16 veces la velocidad de transmisión designada de UART. La señal de muestreo debería funcionar como habilitación de tics en lugar de la señal de reloj al receptor UART.

El generador de velocidad en baudios es un contador programable y el código HDL se muestra en el Listado 4.1.

Listado 4.1. Generador de velocidad

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity baud_gen is
   port(
      clk   : in std_logic;
      reset : in std_logic;
      dvsr  : in std_logic_vector(10 downto 0);
      tick  : out std_logic
   );
end baud_gen;

architecture arch of baud_gen is
   constant N    : integer := 11;
```

```
   signal r_reg  : unsigned(N - 1 downto 0);
   signal r_next : unsigned(N - 1 downto 0);
begin
   -- register
   process(clk, reset)
   begin
      if (reset = '1') then
         r_reg <= (others => '0');
      elsif (clk'event and clk = '1') then
         r_reg <= r_next;
      end if;
   end process;
   -- next-state logic
   r_next <= (others=>'0') when r_reg=unsigned(dvsr) else r_reg + 1;
   -- output logic
   tick <= '1' when r_reg=1 else '0'; -- not use 0 because of reset
end arch;
```

El término `dvsr` (para divisor) es una señal externa y, por lo tanto, su valor se puede configurar dinámicamente durante la operación. Además, `dvsr` (en lugar de `dvsr-1`) se utiliza para reducir la complejidad del hardware. Si el valor de `dvsr` es *v*, el contador cuenta de 0 a v y se ajusta. Por lo tanto, es un contador tipo *(v + 1)*.

El valor v depende de la velocidad de transmisión deseada y la velocidad de reloj del sistema. Deje que la velocidad en baudios *b* y la velocidad de reloj del sistema *f*. La frecuencia de muestreo deseada se convierte en 16 * *b* y el contador debe contar $\frac{f}{16 * b}$ y ajustarse. Significa que

$$\mathrm{v} + 1 = \frac{\mathrm{f}}{16 * \mathrm{b}}$$

y *v* se convierte en $\frac{f}{16 * b} - 1$.

5.4.2.3 Receptor UART

El receptor UART obtiene el byte de datos de la línea serie a través del sobremuestreo. Utiliza el tick del generador de velocidad en baudios para estimar los puntos medios de los bits transmitidos y luego los recupera en estos puntos en consecuencia. La operación de recepción general se puede describir mediante un gráfico ASMD, como se muestra en la Ilustración 26. Para adaptarse a futuras modificaciones, se utilizan dos genéricos en la descripción. DBIT indica el número de bits de datos y `SB_TICK` indica el número de tics necesarios para los bits de parada, que es 16, 24 y 32 para 1, 1.5 y 2 bits de parada, respectivamente.

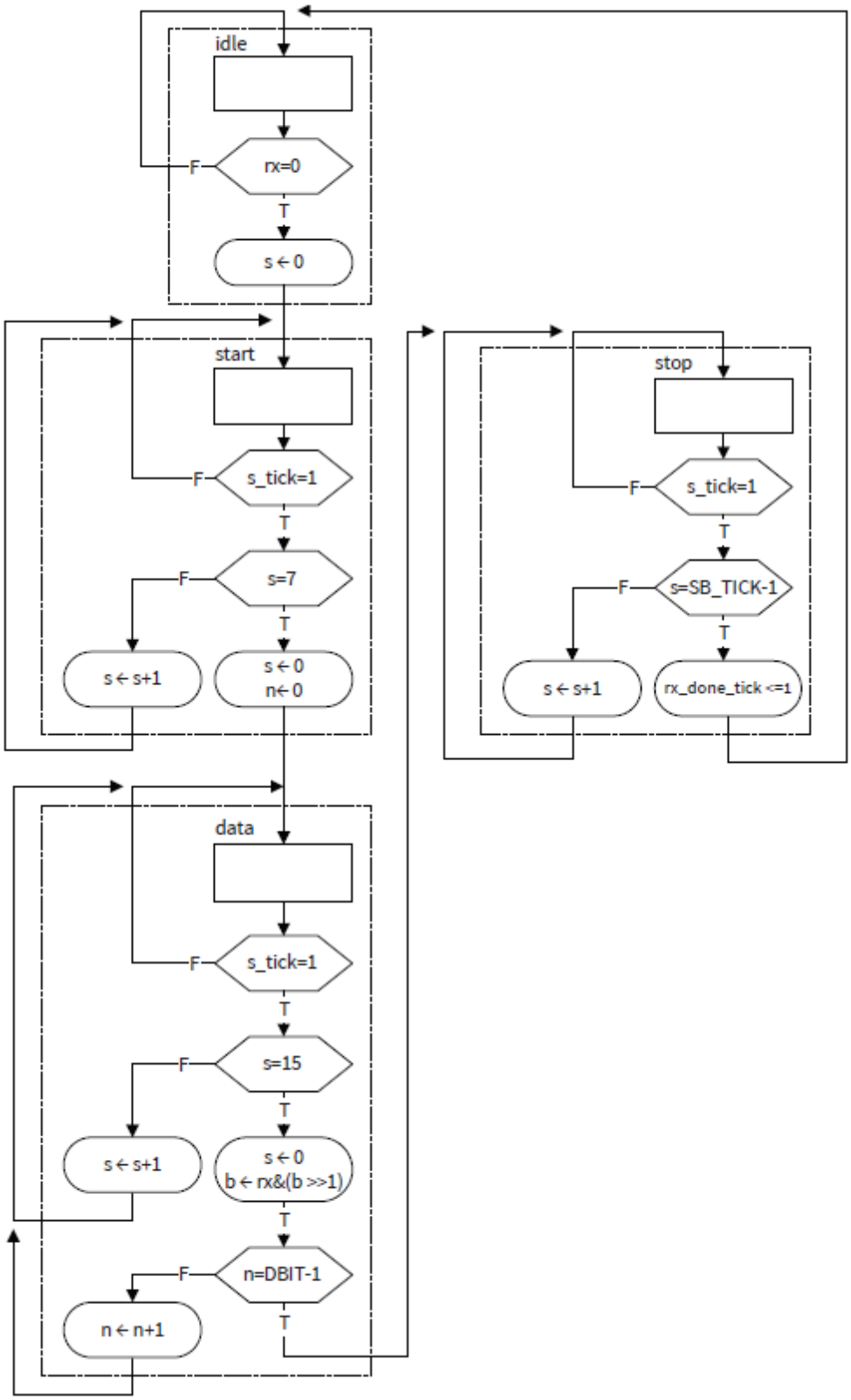


Ilustración 26: *Gráfico ASMD de un receptor UART.*

El gráfico ASMD sigue los pasos descritos en la Sección 5.4.1.3 e incluye tres estados principales, `inicio`, `datos` y `parada`, que representan el procesamiento del bit de inicio, los bits de datos y los bits de parada. La señal `s_tick` es la marca de activación del generador de velocidad en baudios y hay 16 señales en un intervalo de bits. Hay que tener en cuenta que la FSMD permanece en el mismo estado a menos que se confirme la señal `s_tick`. Hay dos contadores, representados por los registros syn. El registro `s` realiza un seguimiento del número de tics de muestreo y cuenta hasta 7 en el estado de `inicio`, hasta 15 en el estado de `datos` y hasta `SB_TICK` en el estado de `parada`. El registro `n` realiza un seguimiento del número de bits de datos recibidos en el estado de datos. Los bits recuperados se desplazan y se vuelven a ensamblar en el registro b. Se incluye una señal de estado, `rx_done_tick`. Se hace valer para un ciclo de reloj después de que se completa el proceso de recepción.

El código correspondiente se muestra en el Listado 4.2. Dado que la señal de `rx` entrante no es controlada por el reloj del sistema, se agrega un sincronizador adicional de dos registros Flip-Flop (FF).

Listado 4.2. Controlador MMIO

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use work.chu_io_map.all;
entity chu_mmio_controller is
   port(
      -- FPro bus
      mmio_cs             : in  std_logic;
      mmio_wr             : in  std_logic;
      mmio_rd             : in  std_logic;
      mmio_addr           : in  std_logic_vector(20 downto 0);
      mmio_wr_data        : in  std_logic_vector(31 downto 0);
      mmio_rd_data        : out std_logic_vector(31 downto 0);
      -- slot interface
      slot_cs_array       : out std_logic_vector(63 downto 0);
      slot_mem_rd_array   : out std_logic_vector(63 downto 0);
      slot_mem_wr_array   : out std_logic_vector(63 downto 0);
      slot_reg_addr_array : out slot_2d_reg_type;
      slot_rd_data_array  : in  slot_2d_data_type;
      slot_wr_data_array  : out slot_2d_data_type
   );
end chu_mmio_controller;

architecture arch of chu_mmio_controller is
   -- 11 LSBs of address used; 2^6 slots, each with 2^5 registers
   alias slot_addr : std_logic_vector(5 downto 0) is
         mmio_addr(10 downto 5);
   alias reg_addr  : std_logic_vector(4 downto 0) is
         mmio_addr(4 downto 0);
begin
   -- address decoding
      process(slot_addr, mmio_cs)
   begin
      slot_cs_array <= (others => '0');
      if mmio_cs = '1' then
         slot_cs_array(to_integer(unsigned(slot_addr))) <= '1';
      end if;
   end process;
```

```
   -- broadcast to all slots
   slot_mem_rd_array   <= (others => mmio_rd);
   slot_mem_wr_array   <= (others => mmio_wr);
   slot_wr_data_array  <= (others => mmio_wr_data);
   slot_reg_addr_array <= (others => reg_addr);
   -- mux for read data
   mmio_rd_data <= slot_rd_data_array(to_integer(unsigned(slot_addr)));
end arch;
```

5.4.2.4 Transmisor UART

El transmisor UART es esencialmente un registro de desplazamiento que desplaza los bits de datos a una velocidad específica. La velocidad se controla mediante la misma señal de activación generada por el generador de velocidad en baudios. Cada bit tiene una duración de 16 ticks. El FSMD del transmisor UART es similar al del receptor UART. Después de confirmar la señal `tx_start`, el FSMD carga la palabra de datos y luego avanza gradualmente a través de los estados de `inicio`, `datos` y `parada` para desplazar los bits correspondientes. Señala la finalización al asegurar la señal `tx_done_tick` para un ciclo de reloj. Se utiliza un búfer de un bit, `tx_reg`, para filtrar cualquier posible error. El código correspondiente se muestra en el Listado 4.3.

Listado 4.3. Transmisor UART

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity uart_tx is
   generic(
      DBIT    : integer := 8;   -- # data bits
      SB_TICK : integer := 16   -- # ticks for stop bits
   );
   port(
      clk, reset   : in  std_logic;
      tx_start     : in  std_logic;
      s_tick       : in  std_logic;
      din          : in  std_logic_vector(7 downto 0);
      tx_done_tick : out std_logic;
      tx           : out std_logic
   );
end uart_tx;

architecture arch of uart_tx is
   type state_type is (idle, start, data, stop);
   signal state_reg       : state_type;
   signal state_next      : state_type;
   signal s_reg, s_next   : unsigned(4 downto 0);
   signal n_reg, n_next   : unsigned(2 downto 0);
   signal b_reg, b_next   : std_logic_vector(7 downto 0);
   signal tx_reg, tx_next : std_logic;
begin
   -- FSMD state & data registers
   process(clk, reset)
   begin
      if reset = '1' then
         state_reg <= idle;
         s_reg     <= (others => '0');
         n_reg     <= (others => '0');
         b_reg     <= (others => '0');
         tx_reg    <= '1';
      elsif (clk'event and clk = '1') then
         state_reg <= state_next;
         s_reg     <= s_next;
         n_reg     <= n_next;
```

```
            b_reg     <= b_next;
            tx_reg    <= tx_next;
         end if;
      end process;
      -- next-state logic & data path
      process(state_reg,s_reg,n_reg,b_reg,s_tick,tx_reg,tx_start,din)
      begin
         state_next   <= state_reg;
         s_next       <= s_reg;
         n_next       <= n_reg;
         b_next       <= b_reg;
         tx_next      <= tx_reg;
         tx_done_tick <= '0';
         case state_reg is
            when idle =>
               tx_next <= '1';
               if tx_start = '1' then
                  state_next <= start;
                  s_next     <= (others => '0');
                  b_next     <= din;
               end if;
            when start =>
               tx_next <= '0';
               if (s_tick = '1') then
                  if s_reg = 15 then
                     state_next <= data;
                     s_next     <= (others => '0');
                     n_next     <= (others => '0');
                  else
                     s_next <= s_reg + 1;
                  end if;
               end if;
            when data =>
               tx_next <= b_reg(0);
               if (s_tick = '1') then
                  if s_reg = 15 then
                     s_next <= (others => '0');
                     b_next <= '0' & b_reg(7 downto 1);
                     if n_reg = (DBIT - 1) then
                        state_next <= stop;
                     else
                        n_next <= n_reg + 1;
                     end if;
                  else
                     s_next <= s_reg + 1;
                  end if;
               end if;
            when stop =>
               tx_next <= '1';
               if (s_tick = '1') then
                  if s_reg = (SB_TICK - 1) then
                     state_next   <= idle;
                     tx_done_tick <= '1';
                  else
                     s_next <= s_reg + 1;
                  end if;
               end if;
         end case;
      end process;
      tx <= tx_reg;
   end arch;
```

5.4.2.5 Códigos HDL de alto nivel

El código HDL de alto nivel sigue el diagrama de la Figura 4.2 y crea una instancia de los cinco componentes principales. El código se muestra en el Listado 4.4.

Listado 4.4. Descripción UART de alto nivel

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity uart is
   generic(
      DBIT    : integer := 8;   -- # data bits
      SB_TICK : integer := 16;  -- # ticks for stop bits, 16 per bit
      FIFO_W  : integer := 4    -- # FIFO addr bits (depth: 2^FIFO_W)
   );
   port(
      clk, reset : in  std_logic;
      rd_uart    : in  std_logic;
      wr_uart    : in  std_logic;
      dvsr       : in  std_logic_vector(10 downto 0);
      rx         : in  std_logic;
      w_data     : in  std_logic_vector(7 downto 0);
      tx_full    : out std_logic;
      rx_empty   : out std_logic;
      r_data     : out std_logic_vector(7 downto 0);
      tx         : out std_logic
   );
end uart;

architecture str_arch of uart is
   signal tick              : std_logic;
   signal rx_done_tick      : std_logic;
   signal tx_fifo_out       : std_logic_vector(7 downto 0);
   signal rx_data_out       : std_logic_vector(7 downto 0);
   signal tx_empty          : std_logic;
   signal tx_fifo_not_empty : std_logic;
   signal tx_done_tick      : std_logic;
begin
   baud_gen_unit : entity work.baud_gen(arch)
      port map(
         clk  => clk, reset => reset, dvsr => dvsr,
         tick => tick
      );
   uart_rx_unit : entity work.uart_rx(arch)
      generic map(DBIT => DBIT, SB_TICK => SB_TICK)
      port map(
         clk          => clk,
         reset        => reset,
         rx           => rx,
         s_tick       => tick,
         rx_done_tick => rx_done_tick,
         dout         => rx_data_out
      );
   uart_tx_unit : entity work.uart_tx(arch)
      generic map(DBIT => DBIT, SB_TICK => SB_TICK)
      port map(
         clk          => clk,
         reset        => reset,
         tx_start     => tx_fifo_not_empty,
         s_tick       => tick,
         din          => tx_fifo_out,
         tx_done_tick => tx_done_tick,
         tx           => tx
      );
   fifo_rx_unit : entity work.fifo(reg_file_arch)
      generic map(DATA_WIDTH => DBIT, ADDR_WIDTH => FIFO_W)
      port map(
         clk    => clk,
         reset  => reset,
         rd     => rd_uart,
         wr     => rx_done_tick,
         w_data => rx_data_out,
         empty  => rx_empty,
         full   => open,
         r_data => r_data
      );
```

```
fifo_tx_unit : entity work.fifo(reg_file_arch)
      generic map(DATA_WIDTH => DBIT, ADDR_WIDTH => FIFO_W)
      port map(
         clk    => clk,
         reset  => reset,
         rd     => tx_done_tick,
         wr     => wr_uart,
         w_data => w_data,
         empty  => tx_empty,
         full   => tx_full,
         r_data => tx_fifo_out
      );
   tx_fifo_not_empty <= not tx_empty;
end str_arch;
```

Los genéricos especifican el número de bits de datos, el número de bits de parada y el tamaño de los buffers FIFO.

5.4.3 DESARROLLO DEL BLOQUE UART

Podemos seguir el procedimiento en la Sección 5.3.3.2 para agregar un circuito de envoltura para el controlador UART y crear un núcleo MMIO IP.

5.4.3.1 Mapa de registro

El procesador interactúa con el controlador UART realizando acciones de la siguiente manera:

- Establecer (escribir) el valor del divisor del generador de velocidad en baudios.
- Recibir (leer) un byte de datos del búfer FIFO revelador.
- Generar (escribir) un impulso para eliminar un byte de datos del búfer FIFO de recepción.
- Verificar (leer) la señal `rx_empty` para determinar si un byte de datos está en el búfer FIFO de recepción.
- Transmitir (escribir) un byte de datos al búfer FIFO de transmisión.
- Verificar (leer) la señal `tx_full` para determinar si el búfer FIFO de transmisión está disponible.

Con base en estas interacciones, podemos definir el mapa de registro del bloque UART. Para mayor claridad, separamos las operaciones de lectura y escritura en diferentes registros. La dirección de desplazamiento y campos del registro de lectura son:

- offset 0 (leer datos y registro de estado)
    - bits 7 a 0: datos de 8 bits recibidos

- bit 8: estado vacío del búfer FIFO de recepción
- bit 9: estado completo del búfer FIFO de transmisión

Hay tres registros de escritura cuyos desplazamientos y campos de dirección son:

- offset 1 (registro de divisor de velocidad en baudios)
  - bits 10 a 0: valor del divisor de 11 bits
- offset 2 (registro de escritura de datos)
  - bits 7 a 0: datos transmitidos de 8 bits
- offset 3 (lectura del registro de eliminación de datos)
  - escritura de datos ficticios que genera un impulso para eliminar un byte de datos del búfer FIFO de recepción

Hay que tener en cuenta que el término "registro" aquí solo se utiliza para expresar una ubicación conceptual dentro del bloque de E/S. Puede o no corresponder a un registro físico.

5.4.3.2 Circuito de envoltura para la interfaz de ranura.

Basados en el mapa de registro y las señales de E/S del controlador UART, se puede derivar un circuito de envoltura que cumpla con la especificación del slot o ranura y crear el bloque UART. El código HDL del bloque se muestra en el Listado 4.5.

Listado 4.5. Bloque UART

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity chu_uart is
   generic(
      FIFO_DEPTH_BIT : integer := 8 -- # FIFO addr bits
   );
   port(
      clk     : in  std_logic;
      reset   : in  std_logic;
      -- slot interface
      cs      : in  std_logic;
      write   : in  std_logic;
      read    : in  std_logic;
      addr    : in  std_logic_vector(4 downto 0);
      rd_data : out std_logic_vector(31 downto 0);
      wr_data : in  std_logic_vector(31 downto 0);
      -- external signals
      tx      : out std_logic;
      rx      : in  std_logic
   );
end chu_uart;
```

```
architecture arch of chu_uart is
   signal wr_en    : std_logic;
   signal wr_uart  : std_logic;
   signal rd_uart  : std_logic;
   signal wr_dvsr  : std_logic;
   signal tx_full  : std_logic;
   signal rx_empty : std_logic;
   signal r_data   : std_logic_vector(7 downto 0);
   signal dvsr_reg : std_logic_vector(10 downto 0);
begin
   -- instantiate uart controller
   uart_unit : entity work.uart(str_arch)
      generic map(
         DBIT    => 8,
         SB_TICK => 16,
         FIFO_W  => FIFO_DEPTH_BIT
      )
      port map(
         clk      => clk,
         reset    => reset,
         rd_uart  => rd_uart,
         wr_uart  => wr_uart,
         dvsr     => dvsr_reg,
         rx       => rx,
         tx       => tx,
         w_data   => wr_data(7 downto 0),
         r_data   => r_data,
         tx_full  => tx_full,
         rx_empty => rx_empty
      );
   -- baud rate register
   process(clk, reset)
   begin
      if (reset = '1') then
         dvsr_reg <= (others => '0');
      elsif (clk'event and clk = '1') then
         if wr_dvsr = '1' then
            dvsr_reg <= wr_data(10 downto 0);
         end if;
      end if;
   end process;
   -- write decoding
   wr_en   <= '1' when write = '1' and cs = '1' else '0';
   wr_dvsr <= '1' when addr(1 downto 0)="01" and wr_en = '1' else '0';
   wr_uart <= '1' when addr(1 downto 0)="10" and wr_en = '1' else '0';
   rd_uart <= '1' when addr(1 downto 0)="11" and wr_en = '1' else '0';
   -- read multiplexing
   rd_data <= x"00000" & "00" & tx_full & rx_empty & r_data;
end arch;
```

El circuito envolvente consta de una instancia de controlador UART, un registro para almacenar el valor del divisor de velocidad en baudios y un circuito de decodificación de escritura. En este caso no es necesaria la multiplexación ya que el controlador UART tiene incorporada la recepción y transmisión de buffers FIFO, así que se crea un registro de E/S para recibir y transmitir datos. Hay que tener en cuenta que los genéricos `DBIT` y `SB_TICK` se asignan a 8 y 16 bits en el diseño, lo que lleva a un UART con ocho bits de datos y un bit de stop, la configuración más ampliamente utilizada.

El circuito de decodificación utiliza los dos LSB (bits de menor peso) de `addr` y `cs` para generar tres señales de habilitación. Las señales `wr_dvsr` y `wr_uart` son usadas para escribir datos en el registro del divisor y en el búfer FIFO de transmisión, respectivamente. La señal `rd_uart` actúa como una marca de eliminación de datos de reloj.

Los datos de lectura se construyen con los datos del búfer FIFO de recepción y dos señales de estado FIFO. Como solo hay un registro de lectura en el mapa de registro, no es necesario realizar multiplexaciones.

Al recibir un elemento de datos del búfer FIFO, se puede eliminar el elemento o mantenerlo intacto. Al mantenerlo intacto se proporciona más flexibilidad para el desarrollo de software, y en el caso de la operación eliminación, se requiere una instrucción separada. Cuando el procesador escribe datos ficticios en el registro de E/S con offset 3, `rd_uart` se valida para un ciclo de reloj y se elimina el byte de datos en la cabecera del receptor.

Alternativamente, `rd_uart` puede estar vinculado a una operación de lectura:

```
rd_uart <= '1' when read = '1' and cs = '1' else '0';
```

El elemento de datos se eliminará automáticamente durante una operación de lectura y no se necesitan instrucciones de escritura por separado. Sin embargo, dado que el búfer FIFO puede estar vacío, es necesario verificar el estado de `rx_empty` para verificar que los datos de lectura sean válidos.

5.4.4 CONTROLADOR DE UART

El controlador UART consta de dos conjuntos de rutinas. El primer conjunto accede a los registros de E/S y realiza operaciones básicas como transmitir un byte, recibir un byte y verificar el estado. El segundo conjunto transmite y muestra una cadena o un número en una consola. Las rutinas pueden considerarse como una versión primitiva de `printf()`. Sin embargo, son muy simples y no requieren mucho espacio de memoria. Si se desean más funciones de formato e impresión, el segundo conjunto se puede separar en una nueva clase con métodos más sofisticados.

5.4.4.1 Definición de clase

La definición de clase del bloque UART se muestra en el Listado 4.6.

Listado 4.6. Definición de clase `UartCore (uart_core.h)`

```
#ifndef _UART_CORE_H_INCLUDED
#define _UART_CORE_H_INCLUDED

#include "chu_io_rw.h"
#include "chu_io_map.h"  // to use SYS_CLK_FREQ
class UartCore {
   /* Register map */
   enum {
      RD_DATA_REG = 0,   /**< rx data/status register */
      DVSR_REG = 1,      /**< baud rate diviser register */
      WR_DATA_REG = 2,   /**< wr data register */
      RM_RD_DATA_REG = 3 /**< remove read data offset */
   };
```

```
 /* mask fields */
   enum {
      TX_FULL_FIELD = 0x00000200, /**< bit 9 of rd_data_reg; full bit  */
      RX_EMPT_FIELD = 0x00000100, /**< bit 10 of rd_data_reg; empty bit */
      RX_DATA_FIELD = 0x000000ff  /**< bits 7..0 rd_data_reg; read data */
   };
public:
   /* methods */
   UartCore(uint32_t core_base_addr);
   ~UartCore();
   //basic I/O access
   void set_baud_rate(int baud);
   int rx_fifo_empty();
   int tx_fifo_full();
   void tx_byte(uint8_t byte);
   int rx_byte();
   //display methods
   void disp(char ch);
   void disp(double f, int digit);
   void disp(double f);
private:
   uint32_t base_addr;
   int baud_rate;
   void disp_str(const char *str);
};
#endif  // _UART_CORE_H_INCLUDED
```

La primera definición de enumeración utiliza nombres simbólicos para los cuatro offsets de registro. La segunda definición de enumeración especifica las máscaras para obtener el byte de datos y los bits de estado del registro `RD_DATA_REG`.

5.4.4.2 Implementación de clase

La implementación de clase del constructor y el primer conjunto de métodos se muestran en el Listado 4.7.

Listado 4.7. Métodos básicos `UartCore` (`uart_core.cpp`)

```
UartCore::UartCore(uint32_t core_base_addr) {
   base_addr = core_base_addr;
   set_baud_rate(9600);      //default baud rate
}

UartCore::~UartCore() {
}

/* baud rate = sys_freq/16/(dvsr+1) */
void UartCore::set_baud_rate(int baud) {
   uint32_t dvsr;

   dvsr = SYS_CLK_FREQ*1000000 / 16 / baud - 1;
   io_write(base_addr, DVSR_REG, dvsr);
}

int UartCore::rx_fifo_empty() {
   uint32_t rd_word;
   int empty;

   rd_word = io_read(base_addr, RD_DATA_REG);
   empty = (int) (rd_word & RX_EMPT_FIELD) >> 8;
   return (empty);
}

int UartCore::tx_fifo_full() {
   uint32_t rd_word;
   int full;
```

```
   rd_word = io_read(base_addr, RD_DATA_REG);
   full = (int) (rd_word & TX_FULL_FIELD) >> 9;
   return (full);
}

void UartCore::tx_byte(uint8_t byte) {
   while (tx_fifo_full()) {
   };  // busy waiting
   io_write(base_addr, WR_DATA_REG, (uint32_t )byte);
}

int UartCore::rx_byte() {
   uint32_t data;

   if (rx_fifo_empty())
      return (-1);
   else {
      data = io_read(base_addr, RD_DATA_REG) & RX_DATA_FIELD;
      io_write(base_addr, RM_RD_DATA_REG, 0); //dummy write to remove data from rx FIFO
      return ((int) data);
   }
}
```

El constructor `UartCore()` guarda la dirección base y establece la velocidad en baudios en un valor predeterminado de 9600. El método `set_baud_rate()` calcula el valor del divisor a la velocidad deseada y lo escribe en el registro del divisor. Este método puede invocarse si la velocidad deseada es diferente del valor predeterminado. Los métodos `rx_fifo_empty()` y `tx_fifo_full()` extraen y devuelven el bit de estado correspondiente.

El método `tx_byte()` transmite un byte de datos. "Espera ocupado" hasta que un espacio en el búfer FIFO de transmisión esté disponible y luego escribe los datos en el búfer. El método `rx_byte()` intenta recuperar un byte de datos. Primero verifica si el búfer FIFO de recepción está vacío y devuelve -1 como si la condición fuera verdadera. De lo contrario, recupera el byte de datos y lo elimina del búfer FIFO.

5.4.4.3 Código ASCII

Una de las aplicaciónes principales de un bloque UART es comunicarse con un dispositivo externo y mostrar la información relevante en una consola de comandos. La información se transmite como caracteres en código ASCII, que tiene 7 bits de ancho y consta de 128 palabras de código, incluidos alfabetos regulares, dígitos, símbolos de puntuación y caracteres de control no imprimibles. Los caracteres y sus palabras clave (en formato hexadecimal) se muestran en la Tabla 2. Los caracteres no imprimibles se muestran entre paréntesis, como (del). Varios caracteres no imprimibles pueden introducir una acción especial cuando se reciben:

- (nulo): byte nulo, que es el patrón todo-cero
- (bel): genera un sonido de campana, si es compatible
- (bs): retroceso

- (ht): pestaña horizontal
- (nl): nueva línea
- (vt): pestaña vertical
- (np): nueva página
- (cr): retorno de carro
- (esc): escapar
- (sp): espacio
- (del): delete, que también es el patrón todo en uno

Code	Char	Code	Char	Code	Char	Code	Char
00	(nul)	20	(sp)	40	@	60	‘
01	(soh)	21	!	41	A	61	a
02	(stx)	22	”	42	B	62	b
03	(etx)	23	#	43	C	63	c
04	(eot)	24	$	44	D	64	d
05	(enq)	25	%	45	E	65	e
06	(ack)	26	&	46	F	66	f
07	(bel)	27	’	47	G	67	g
08	(bs)	28	(	48	H	68	h
09	(ht)	29	)	49	I	69	i
0a	(nl)	2a	*	4a	J	6a	j
0b	(vt)	2b	+	4b	K	6b	k
0c	(np)	2c	,	4c	L	6c	l
0d	(cr)	2d	-	4d	M	6d	m
0e	(so)	2e	.	4e	N	6e	n
0f	(si)	2f	/	4f	O	6f	o
10	(dle)	30	0	50	P	70	p
11	(dc1)	31	1	51	Q	71	q
12	(dc2)	32	2	52	R	72	r
13	(dc3)	33	3	53	S	73	s
14	(dc4)	34	4	54	T	74	t
15	(nak)	35	5	55	U	75	u
16	(syn)	36	6	56	V	76	v
17	(etb)	37	7	57	W	77	w
18	(can)	38	8	58	X	78	x
19	(em)	39	9	59	Y	79	y
1a	(sub)	3a	:	5a	Z	7a	z
1b	(esc)	3b	;	5b	[	7b	{
1c	(fs)	3c	¡	5c	\	7c	—
1d	(gs)	3d	=	5d	]	7d	}
1e	(rs)	3e	¿	5e	^	7e	~
1f	(us)	3f	?	5f	_	7f	(del)

Tabla 2: *Códigos ASCII.*

Muchas funcionalidades de controlador dedicado implican comunicación en serie. Las siguientes observaciones nos ayudan a manipular y procesar el código ASCII:

- Cuando el primer dígito hexadecimal en una palabra establecida en código es 0x0 o 0x1, el carácter correspondiente es un carácter de control.
- Cuando el primer dígito hexadecimal en una palabra clave es 0x2 o 0x3, el carácter correspondiente es un dígito o puntuación.
- Cuando el primer dígito hexadecimal en una palabra de código es 0x4 o 0x5, el carácter correspondiente generalmente es una letra mayúscula.
- Cuando el primer dígito hexadecimal en una palabra clave es 0x6 o 0x7, el carácter correspondiente es generalmente una letra minúscula.
- Si el primer dígito hexadecimal en una palabra clave es 0x3, el segundo dígito hexadecimal representa el dígito decimal correspondiente.
- Las letras mayúsculas y minúsculas difieren en un solo bit y se pueden convertir unas a otras sumando o restando 0x20 o invirtiendo el sexto bit.

Tenga en cuenta que el código ASCII utiliza solo 7 bits, pero una palabra de datos normalmente se compone de 8 bits (un byte).

5.4.4.4 Métodos de visualización

La implementación de clase del segundo conjunto de rutinas se muestra en el Listado 4.8. Son métodos sobrecargados para transmitir y mostrar una cadena y un número. La función, `disp_str()`, se utiliza para facilitar el procesamiento.

Listado 4.8. Métodos de visualización `UartCore (uart_core.cpp)`

```
void UartCore::disp_str(const char *str) {
   while ((uint8_t) *str) {
      tx_byte(*str);
      str++;
   }
}
void UartCore::disp(const char *str) {
   disp_str(str);
}

void UartCore::disp(char ch) {
    tx_byte(ch);
}

void UartCore::disp(int n, int base, int len) {
   char buf[33];         // 32 bit #
   char *str, ch, sign;
   int rem, i;
   unsigned int un;
   /* error check */
   if (base != 2 && base != 8 && base != 16)
      base = 10;
   if (len > 32)  // error check
```

```
      len = 32;
   /* handle neg decimal # */
   if (base == 10 && n < 0) {
      un = (unsigned) -n;
      sign = '-';
   } else {
      un = (unsigned) n; // interpreted as unsigned for hex/bin conversion
      sign = ' ';
   }
   /* convert # to string */
   str = &buf[33];
   *str = '\0';
   i = 0;
   do {
      str--;
      rem = un % base;
      un = un / base;
      if (rem < 10)
         ch = (char) rem + '0';
      else
         ch = (char) rem - 10 + 'a';
      *str = ch;
      i++;
   } while (un);
   /* attach - sign for neg decimal # */
   if (sign == '-') {
      str--;
      *str = sign;
      i++;
   }
   /* pad with blank */
   while (i < len) {
      str--;
      *str = ' ';
      i++;
   };
   disp_str(str);
}
void UartCore::disp(int n) {
   disp(n, 10, 0);
}
void UartCore::disp(int n, int base) {
   disp(n, base, 0);
}
void UartCore::disp(double f, int digit) {
   double fa, frac; // absolute value of f
   int n, i, i_part;

   fa = f;
   if (f < 0.0) {
      fa = -f;
      disp_str("-");
   }
   // display integer portion
   i_part = (int) fa; // integer part of f
   disp(i_part);
   disp_str(".");
   // display fraction part
   frac = fa - (double) i_part;
   for (n = 0; n < digit; n++) {
    frac = frac * 10.0;
      i = (int) frac;
      disp(i);
      frac = frac - i;
   }
}
void UartCore::disp(double f) {
   disp(f, 3);
}
```

El método `disp_str()` transmite una cadena de caractereres de una vez, que se espera que se muestre en una consola. Los métodos `disp()` sobrecargados

muestran un carácter, una cadena o un número. El `disp(char ch)` transmite `ch` como un byte en bruto sin procesamiento. El método `disp(const char * str)` simplemente llama a `disp_str()`. El método `disp(int n, int base, int len)` convierte un número `n`, en una cadena visualizable. El parámetro `base` especifica la base, que puede ser 2 (para binario), 8 (para octal), 10 (para decimal) o 16 (para hexadecimal). El parámetro `len` especifica el número de dígitos en la cadena (longitud). Los ceros (0) extra se rellenarán delante según sea necesario. No se agregarán rellenos de ceros si `len` es 0. Los otros dos métodos `disp()` relevantes no usan rellenos de ceros y usan representación de base 10. El método `disp(double f, int digit)` convierte un número de coma flotante `f`, en una cadena visualizable. La cadena contiene una parte entera y una parte de fracción. El parámetro `dígit` especifica el número de dígitos en la porción de fracción. El otro método relevante establece que `dígit` sea 3.

5.4.4.5 Prueba

El programa de prueba que se muestra en el Listado 2.11 incluye una función simple, `uart_chec()`, para verificar el funcionamiento del núcleo y controlador UART.

5.5 XILINX XADC

Un ADC (convertidor de analógico a digital) es un circuito que digitaliza un sensor analógico continuo al convertir su nivel de voltaje en una cantidad digital discreta. Los dispositivos FPGA de la serie Xilinx 7 contienen una macro celda, XADC, que proporciona una funcionalidad básica de conversión de analógico a digital. El aspecto de la composición de E/S de un módulo XADC se muestra a continuación:

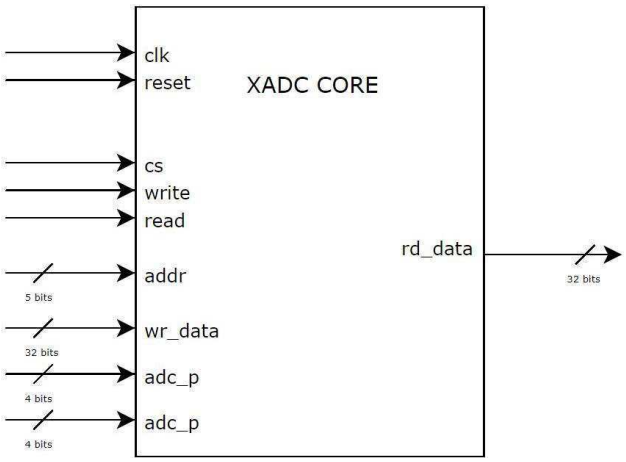


Ilustración 27: *Composición de E/S bloque XADC*

5.5.1 DESCRIPCIÓN GENERAL DE XADC

5.5.1.1 Diagrama de bloques

El diagrama de bloques conceptual de una macro celda XADC se muestra en la Ilustración 28. Se compone de cinco partes principales:

- ADCs duales
- Sensores on-chip y alarma.
- Multiplexores analógicos.
- Registro de control y registro de estado.
- Interfaz DRP (puerto de reconfiguración dinámica)

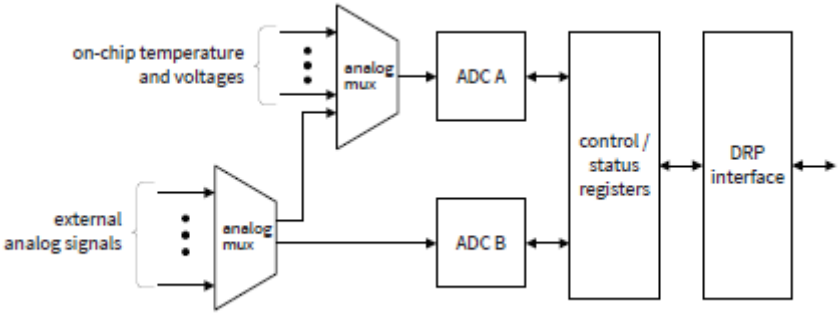


Ilustración 28: *Diagrama de bloques conceptual de XADC.*

Los componentes clave son los dos ADCs. El ADC tiene una resolución de 12 bits y una tasa de muestreo de 100M SPS (Samples Per Second, traducido como muestras por segundo). Utiliza una estructura de entrada diferencial, en la cual la diferencia de voltaje entre dos puntos (las entradas más (+) y menos(-)) se usa para la medición, y la máxima oscilación de la señal de entrada es de 1,0V. La entrada se puede configurar como modo unipolar entre 0,0V y + 1,0V o como modo bipolar entre -0,5V y + 0,5V.

Los dispositivos modernos de FPGA imponen restricciones estrictas en las fuentes de alimentación.

Un multiplexor analógico selecciona una señal de entrada específica y la conecta a la salida. Permite que múltiples canales de entrada analógica se enruten a un solo ADC. Una aplicación útil es compartir un ADC rápido con múltiples canales de baja velocidad de datos en forma de turnos, lo que se conoce como modo secuenciado. Por ejemplo, se pueden multiplexar cuatro canales de entrada a través de un ADC SPS de 100M y la medición y la conversión se realizan sucesivamente. Cada canal de entrada se muestrea a una velocidad de 25M SPS y el sistema aparece como cuatro ADC de 25M SPS. En el modo de secuenciador, se pueden seleccionar los canales necesarios y los sensores On-Chip. Los pines de E/S para los canales analógicos auxiliares no utilizados pueden configurarse como pines de E/S digitales regulares.

El XADC contiene 64 registros de control de 16 bits y 64 registros de estado de 16 bits. Los registros de control almacenan la información de configuración, como la selección de canales. Los registros de estado contienen los resultados de conversión (datos digitalizados) de los canales de entrada y los sensores On-Chip. El rango de direcciones de 0x00 a 0x1f se usa para los resultados de la conversión.

La interfaz DRP (Dynamic Reconfiguration Port o puerto de reconfiguración dinámica) proporciona una interfaz para acceder a los registros internos. Su

especificación básica es similar al bus de E/S de MicroBlaze MCS descrito en la Sección 5.3.6.2.

Además, el XADC proporciona una colección de señales de estado que indican el canal analógico utilizado en la conversión y la finalización de la conversión.

5.5.1.2 Configuración del bloque XADC

El XADC es versátil y flexible y se pueden configurar muchos aspectos. La información de configuración se almacena en sus registros de control. La configuración de "encendido" se puede especificar a través de los valores iniciales del registro. Se puede incluir un circuito de interfaz de escritura y la XADC se puede reconfigurar en tiempo real escribiendo nuevos valores en los registros de control.

5.5.2 DESARROLLO DEL BLOQUE XADC

La placa Nexys 4 DDR utiliza cuatro canales analógicos (canales 2, 3, 10 y 11) y los conecta a un PMOD designado JXADC. Se desarrolla un núcleo o bloque FPro XADC que aparece como cuatro independientes ADCs que muestrean cuatro canales analógicos continuamente. Adicionalmente, el bloque también proporciona las lecturas de temperatura de la matriz y el voltaje del bloque. El desarrollo implica la creación de una instancia XADC debidamente configurada y el diseño de un circuito de envoltura para recuperar las lecturas.

5.5.2.1 Instanciación XADC

Una instancia de XADC debe configurarse de la siguiente manera:

- Configurar la selección de canales y la operación en el modo de secuenciador a multilpexar múltiples canales y sensores On - Chip.
- Configurar el funcionamiento del secuenciador en modo continuo para realizar la conversión de forma continua.
- Configurar el tiempo en continuo para activar automáticamente la conversión.
- Seleccionar la temperatura de la matriz On - Chip y los sensores de voltaje del bloque y los canales analógicos 2, 3, 10 y 11 para la conversión.
- Seleccionar el modo unipolar para el rango de entrada de 0,0V a 1,0V.

La declaración de entidad de la instancia de XADC configurada es

```
entity xadc_fpro is
   port(
      -- clock and reset
      dclk_in     : in  std_logic;
      reset_in    : in  std_logic;
```

```
 -- DRP interface
      daddr_in    : in  std_logic_vector(6 downto 0);
      den_in      : in  std_logic;
      di_in       : in  std_logic_vector(15 downto 0);
      dwe_in      : in  std_logic;
      do_out      : out std_logic_vector(15 downto 0);
      drdy_out    : out std_logic;
      -- auxiliary analog input channel
      vauxp2      : in  std_logic;
      vauxn2      : in  std_logic;
      vauxp3      : in  std_logic;
      vauxn3      : in  std_logic;
      vauxp10     : in  std_logic;
      vauxn10     : in  std_logic;
      vauxp11     : in  std_logic;
      vauxn11     : in  std_logic;
      -- conversion status signals
      busy_out    : out std_logic;
      channel_out : out std_logic_vector(4 downto 0);
      eoc_out     : out std_logic;
      eos_out     : out std_logic;
   );
end xadc_fpro;
```

Las señales son para la interfaz DRP, los cuatro canales de entrada analógica auxiliar y el estado de conversión XADC.

5.5.2.2 Diseño de circuito de envoltura básico

El XADC instanciado está configurado para ejecutarse automáticamente. Mide y convierte las dos lecturas del sensor en el chip y cuatro canales analógicos sucesivamente, y muestra los resultados convertidos más recientemente en los registros de estado correspondientes. La principal funcionalidad del circuito de envoltura es leer el registro de estado designado a través de la interfaz DRP.

La operación de lectura se realiza de la siguiente manera:

- El circuito de control externo coloca la dirección de registro en `daddr_in`.
- El circuito de control externo establece la señal de habilitación de registro, `den_in`, en 1 y establece la señal de habilitación de escritura, `dwe_in`, en 0 (no escritura).
- El XADC recupera los datos del registro designado.
- Cuando se completa la operación, el XADC coloca los datos en `do_out` y afirma `drdy_out` para indicar que los datos están listos.

La operación de lectura de DRP puede tomar múltiples ciclos de reloj y las funciones `drdy_out` como un reconocimiento. Esto no es compatible con la especificación del bus FPro, en el que la lectura se completa en un ciclo de reloj.

Una forma de diseñar el circuito de envoltura es acceder directamente a la interfaz DRP. El circuito incluirá un registro de bandera asociado con `drdy_out`

y el controlador del software sondea el registro para verificar la validez de los datos de lectura.

Para simplificar la interfaz y el controlador, se utiliza un circuito de envoltura alternativo. El diseño contiene una colección de registros externos para mantener las lecturas digitalizadas y utiliza la señal de estado XADC para actualizar sus contenidos. El diagrama de bloques se muestra en la Ilustración 29. La operación se realiza en dos fases. En la primera fase, la señal de estado de finalización XDAC activa una operación de lectura para recuperar los datos recién convertidos. En la segunda fase, la señal de listo del DRP habilita y almacena los datos en el registro externo designado.

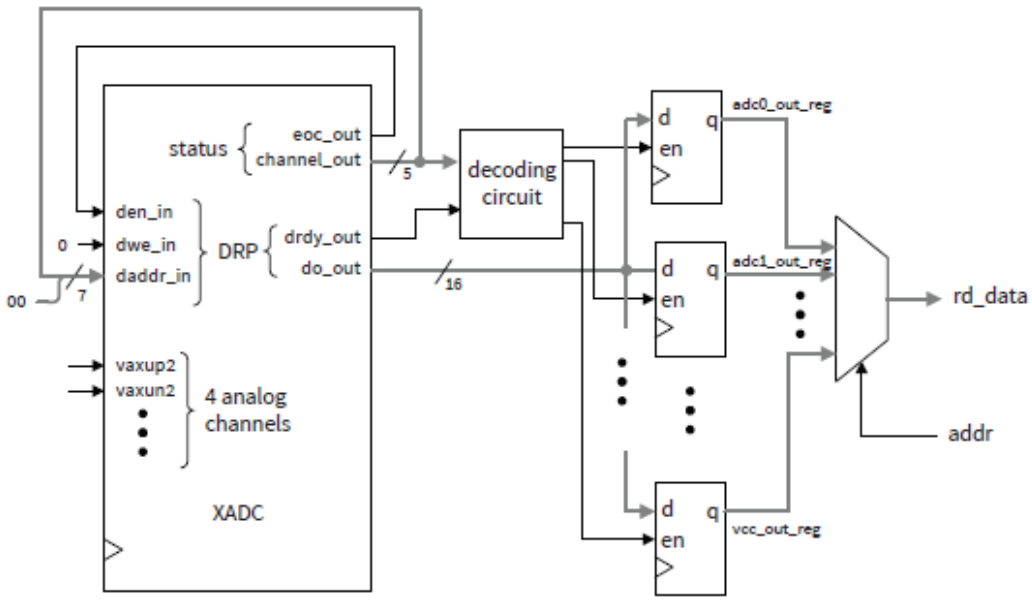


Ilustración 29: *Diagrama de bloques del circuito de envoltura del bloque XADC.*

Cuando se completa una conversión, el bloque de XADC almacena el resultado en el registro de estado correspondiente. Al mismo tiempo, coloca el número de canal, que es el mismo que los cinco LSB (bits menos significativos) de la dirección del registro de estado, en `channel_out`, y afirma la señal `eoc_out` (para el "fin de la conversión") para un ciclo de reloj. Se utiliza la señal `eoc_out` como la señal de habilitación de DRP y el número de canal extendido como la dirección de DRP para iniciar una operación de lectura, que recupera los datos recién convertidos.

Cuando DRP completa la operación de lectura, coloca los datos en `do_out` y confirma la señal de listo, `drdy_out` para un ciclo de reloj. La señal `drdy_out`

junto con el número de canal habilita y almacena los datos recién convertidos en el registro externo designado.

En resumen, el circuito de envoltura aparece como seis registros que mantienen las lecturas actuales de dos sensores On -Chip y cuatro entradas analógicas. Los datos se actualizan de forma continua y automática. Un circuito de multiplexación de lectura encamina la lectura seleccionada a la salida.

5.5.2.3 Mapa de registro

El bloque XADC FPro es una colección de seis registros que suministran lecturas de ADC. Organizamos los offsets o desplazamiento de direcciones de registro de la siguiente manera:

- offset 0 a 3: entradas analógicas 0 a 3, que corresponden a los canales de entrada analógica auxiliar XADC 3, 10, 2 y 11, respectivamente.
- offset 4: lectura de temperatura de la matriz On – Chip.
- offset 5: lectura de voltaje interno del bloque On – Chip.

Todos los registros tienen 16 bits de ancho y los 12 MSB son las lecturas de ADC.

5.5.2.4 Código HDL

El código HDL del bloque XADC sigue el diagrama de la Ilustración 29 y se muestra en el Listado 5.1.

Listado 5.1. Bloque XADC

```
library ieee;
use ieee.std_logic_1164.all;
entity chu_xadc_core is
   port(
      clk     : in  std_logic;
      reset   : in  std_logic;
      -- slot interface
      cs      : in  std_logic;
      write   : in  std_logic;
      read    : in  std_logic;
      addr    : in  std_logic_vector(4 downto 0);
      rd_data : out std_logic_vector(31 downto 0);
      wr_data : in  std_logic_vector(31 downto 0);
      -- external signals
      adc_p   : in  std_logic_vector(3 downto 0);
      adc_n   : in  std_logic_vector(3 downto 0)
   );
end chu_xadc_core;

architecture arch of chu_xadc_core is
   component xadc_fpro
      port(
         di_in       : in  std_logic_vector(15 downto 0);
         daddr_in    : in  std_logic_vector(6 downto 0);
         den_in      : in  std_logic;
         dwe_in      : in  std_logic;
         drdy_out    : out std_logic;
```

```
        do_out      : out std_logic_vector(15 downto 0);
        dclk_in     : in  std_logic;
        reset_in    : in  std_logic;
        vp_in       : in  std_logic;
        vn_in       : in  std_logic;
        vauxp2      : in  std_logic;
        vauxn2      : in  std_logic;
        vauxp3      : in  std_logic;
        vauxn3      : in  std_logic;
        vauxp10     : in  std_logic;
        vauxn10     : in  std_logic;
        vauxp11     : in  std_logic;
        vauxn11     : in  std_logic;
        channel_out : out std_logic_vector(4 downto 0);
        eoc_out     : out std_logic;
        alarm_out   : out std_logic;
        eos_out     : out std_logic;
        busy_out    : out std_logic
     );
  end component;
  signal channel      : std_logic_vector(4 downto 0);
  signal daddr_in     : std_logic_vector(6 downto 0);
  signal eoc          : std_logic;
  signal rdy          : std_logic;
  signal adc_data     : std_logic_vector(15 downto 0);
  signal adc0_out_reg : std_logic_vector(15 downto 0);
  signal adc1_out_reg : std_logic_vector(15 downto 0);
  signal adc2_out_reg : std_logic_vector(15 downto 0);
  signal adc3_out_reg : std_logic_vector(15 downto 0);
  signal tmp_out_reg  : std_logic_vector(15 downto 0);
  signal vcc_out_reg  : std_logic_vector(15 downto 0);
begin
  -- instantiate cusomized xadc core
  xdac_unit : xadc_fpro
     port map(
        dclk_in     => clk,
        reset_in    => reset,          --reset,
        di_in       => (others => '0'),
        daddr_in    => daddr_in,
        den_in      => eoc,
        dwe_in      => '0',            -- read only
        drdy_out    => rdy,
        do_out      => adc_data,
        vp_in       => '0',
        vn_in       => '0',
        vauxp2      => adc_p(2),
        vauxn2      => adc_n(2),
        vauxp3      => adc_p(0),
        vauxn3      => adc_n(0),
        vauxp10     => adc_p(1),
        vauxn10     => adc_n(1),
        vauxp11     => adc_p(3),
        vauxn11     => adc_n(3),
        channel_out => channel,
        eoc_out     => eoc,
        eos_out     => open,
        busy_out    => open,
        alarm_out   => open
     );
  -- form xadc DRP address
  daddr_in <= "00" & channel;
  -- registers and decoding
  process(clk, reset)
  begin
     if reset = '1' then
        adc0_out_reg <= (others => '0');
        adc1_out_reg <= (others => '0');
        adc2_out_reg <= (others => '0');
        adc3_out_reg <= (others => '0');
        tmp_out_reg  <= (others => '0');
        vcc_out_reg  <= (others => '0');
     elsif (clk'event and clk = '1') then
        if rdy = '1' and channel = "10011" then
           adc0_out_reg <= adc_data;
```

```
         end if;
         if rdy = '1' and channel = "11010" then
            adc1_out_reg <= adc_data;
         end if;
         if rdy = '1' and channel = "10010" then
            adc2_out_reg <= adc_data;
         end if;
         if rdy = '1' and channel = "11011" then
            adc3_out_reg <= adc_data;
         end if;
         if rdy = '1' and channel = "00000" then
            tmp_out_reg <= adc_data;
         end if;
         if rdy = '1' and channel = "00001" then
            vcc_out_reg <= adc_data;
         end if;
      end if;
   end process;
   -- read multiplexing
   with addr(2 downto 0) select
      rd_data <=
         x"0000" & adc0_out_reg when "000",
         x"0000" & adc1_out_reg when "001",
         x"0000" & adc2_out_reg when "010",
         x"0000" & adc3_out_reg when "011",
         x"0000" & tmp_out_reg  when "100",
         x"0000" & vcc_out_reg  when others;
end arch;
```

Hay que tener en cuenta que XADC tiene 128 registros con un rango de direcciones entre 0x00 y 0x7f y que las lecturas de ADC se asignan a los registros 0x00 a 0x1f. Las lecturas de los 16 canales de entrada analógica auxiliar se almacenan en 0x10 a 0x1f, y las lecturas de la temperatura de la matriz y del voltaje del bloque se almacenan en 0x00 y 0x01, respectivamente.

El extraño mapeo entre los números de canales auxiliares originales de XADC y los "números de canales lógicos" del bloque de FPro se debe al diseño de la placa Nexys 4 DDR. La disposición física de los pines del puerto PMOD de JXADC y la convención de denominación se muestran en la Ilustración 30.

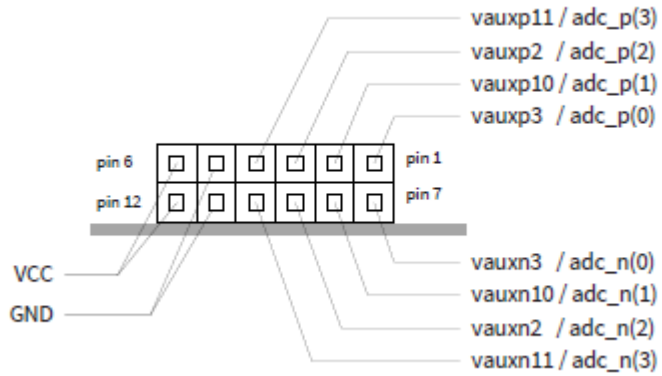


Ilustración 30: *Disposición de pines de entrada analógica del puerto Nexys 4 DDR JXADC PMOD.*

5.5.3 CONTROLADOR DEL BLOQUE DEL DISPOSITIVO XADC

El controlador recupera las seis mediciones de ADC, incluida la temperatura de la matriz y el voltaje del bloque del dispositivo FPGA y los voltajes de los cuatro canales externos.

5.5.3.1 Definición de clase

La definición de clase del bloque XADC se muestra en el Listado 5.2.

Listado 5.2. Definición de clase `XadcCore` (`xadc_core.h`)

```
#ifndef _XADC_CORE_H_INCLUDED
#define _XADC_CORE_H_INCLUDED

#include "chu_init.h"
class XadcCore {
public:
   /* Register map */
   enum {
      ADC_0_REG = 0, /**< 16-bit data from Nexys-4 adc input #0   */
      TMP_REG   = 4,   /**< FPGA internal temperature */
      VCC_REG   = 5,   /**< FPGA internal core volatge */
   };

   /* Constructor */
   XadcCore(uint32_t core_base_addr);
   ~XadcCore(); // not used
   uint16_t read_raw(int n);
   double read_adc_in(int n);
   double read_fpga_vcc();
   double read_fpga_temp();
private:
   uint32_t base_addr;
};
```

La primera definición de `enum` utiliza nombres simbólicos para los desplazaminetos de registro del primer canal de entrada analógica, la temperatura de la matriz en el chip y el registro de voltaje del bloque.

5.5.3.2 Implementación de clase

La implementación de clase del constructor y los métodos se muestran en el Listado 5.3.

Listado 5.3. Implementación de clase `XadcCore` (`xadc_core.cpp`)

```
XadcCore::XadcCore(uint32_t core_base_addr) {
   base_addr = core_base_addr;
}

XadcCore::~XadcCore() {
}

uint16_t XadcCore::read_raw(int n) {
   uint16_t rd_data;

   rd_data = (uint16_t) io_read(base_addr, n) & 0x0000ffff;
   return (rd_data);
}
```

```
double XadcCore::read_adc_in(int n) {
   uint16_t raw;
   raw = read_raw(n) >> 4;
   return ((double) raw / 4096.0);
}

// input source 5 is connected to vcc reading
double XadcCore::read_fpga_vcc() {
   return (read_adc_in(VCC_REG) * 3.0);
}

// input source 4 is connected to temperature reading
double XadcCore::read_fpga_temp() {
   return (read_adc_in(TMP_REG) * 503.975 - 273.15);
```

El constructor `XadcCore()` guarda la dirección base y los otros métodos recuperan y procesan las lecturas de ADC. El método `read_raw()` recupera y devuelve los datos sin procesar de 16 bits, en los que los 12 MSB son las lecturas de ADC con un rango entre 0x000 y 0xfff, que para un canal analógico corresponde al rango entre 0.0V y 1.0V. El método `read_adc_in()` convierte la representación sin signo de los 12 bits a voltaje. La lectura de voltaje del sistema On - Chip se escala a un tercio del valor real. El método `read_fpga_vcc()` realiza el ajuste y devuelve el voltaje real. La relación entre la temperatura de la matriz y la lectura de voltaje es

*Temperatura (ºC) = voltaje * 503.975 - 273.15*

El método `read_fpga_temp()` calcula y devuelve la temperatura.

5.5.3.3 Pruebas para el núcleo XADC

El divisor de voltaje está compuesto por una resistencia de 22K (Ω) y un potenciómetro de 10K (Ω), como se muestra en la Ilustración 31. Los niveles de voltaje mínimo y máximo que cruzan el potenciómetro son 0.00V $\left(\frac{0K}{22K+10K} * 3.3V\right)$ y 1.03V $\left(\frac{10K}{22K+10K} * 3.3V\right)$, que proporcionan el rango deseado. El potenciómetro se puede considerar como otra forma de dispositivo de entrada que proporciona un ajuste de incremento sobre unos 4000 valores.

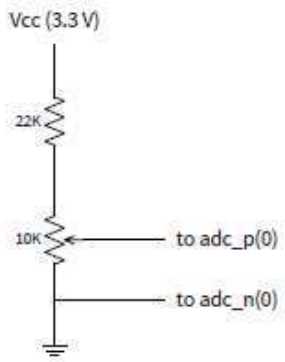


Ilustración 31: *Divisor de tensión para una entrada analógica.*

Se utiliza una función de prueba simple para verificar el funcionamiento del XADC core, como se muestra en el Listado 5.4.

Listado 5.4. Función de prueba XADC (`main_sampler_test.cpp`)

```
void adc_check(XadcCore *adc_p, GpoCore *led_p) {
   double reading;
   int n, i;
   uint16_t raw;

   for (i = 0; i < 5; i++) {
      // display 12-bit channel 0 reading in LED
      raw = adc_p->read_raw(0);
      raw = raw >> 4;
      led_p->write(raw);
      // display on-chip sensor and 4 channels in console
      uart.disp("FPGA vcc/temp: ");
      reading = adc_p->read_fpga_vcc();
      uart.disp(reading, 3);
      uart.disp(" / ");
      reading = adc_p->read_fpga_temp();
      uart.disp(reading, 3);
      uart.disp("\n\r");
      for (n = 0; n < 4; n++) {
         uart.disp("analog channel/voltage: ");
         uart.disp(n);
         uart.disp(" / ");
         reading = adc_p->read_adc_in(n);
         uart.disp(reading, 3);
         uart.disp("\n\r");
      } // end for
      sleep_ms(200);
   }
}
```

5.5.4 SISTEMA FPRO DE PRUEBA

Después de desarrollar un bloque, debe integrarse en un sistema FPro y probarse. Se crea un sistema que incorpora todos estos bloques y lo llamamos un sistema de muestreo FPro. El procedimiento de prueba general y el sistema de muestreo se discuten en las siguientes subsecciones.

5.5.4.1 Procedimiento de prueba de un bloque FPro

A continuación se presenta el procedimiento general para verificar el funcionamiento de un bloque FPro MMIO:

- Asignar un nuevo número de slot o ranura para la instancia central. Registrar el número como una constante simbólica en los archivos `chu_io_map.h` y `chu_io_map.vhd`, como se explica en la sección 5.2.4.1.
- Conectar el núcleo a la ranura designada del controlador MMIO y crear un nuevo subsistema MMIO, similar al de la sección 5.3.5.3.
- Construir un nuevo sistema FPro que incorpore los subsistemas MMIO, similar al de la sección 5.3.7, y realizar una síntesis, ubicación y enrutamiento.

- Derivar el controlador de dispositivo para el bloque, como se explica en la sección 5.2.6.

- Desarrollar un programa de verificación de software y obtener el archivo *.elf*, como se explica en la sección 5.5.4.4.

- Regenerar el archivo de bits, programar el dispositivo FPGA y verificar la operación física del bloque.

5.5.4.2 Configuración del sistema

Las asignaciones de espacios y los nombres simbólicos correspondientes se muestran en la Tabla 3. Los nombres se utilizan en los archivos `chu_io_map.vhd` y `chu_io_map.h`. Los primeros cuatro slots o ranuras son para los cuatro bloques básicos, idénticos al sistema vanilla FPro en la Sección 5.3.7. La ranura 4 está reservada para facilitar la creación de nuevos prototipos de bloque. Se puede insertar un nuevo core definido por el usuario en esta ranura sin afectar la configuración del muestreador. Las ranuras 5 a 13 se pueden utilizar para añadir nueve cores como los mostrados en la Tabla 3:

Slot	Symbolic name	Description
0	S0_SYS_TIMER	system timer core
1	S1_UART1	UART core #1
2	S2_LED	GPO core for discrete LEDs
3	S3_SW	GPI core for discrete switches
4	S4_USER	user prototyping core
5	S5_XADC	Xilinx ADC core
6	S6_PWM	pulse width modulation core
7	S7_BTN	debouncing core for buttons
8	S8_SSEG	LED multiplexing core for seven-segment LED display
9	S9_SPI	SPI core
10	S10_I2C	I2C core
11	S11_PS2	PS2 core for keyboard or mouse
12	S12_DDFS	direct digital frequency synthesis core
13	S13_ADSR	ADSR envelope generator core

Tabla 3: *Asignación de ranuras.*

5.5.4.3 Derivación de hardware

El código del subsistema MMIO se muestra en el Listado 5.5. Los bloques MMIO son definidos en función de la asignación de ranuras en la Tabla 3.

Listado 5.5. Subsistema de prueba MMIO

```
library ieee;
use ieee.std_logic_1164.all;
use work.chu_io_map.all;
```

```
entity mmio_sys_sampler is
   port(
      -- FPro bus
      clk          : in    std_logic;
      reset        : in    std_logic;
      mmio_cs      : in    std_logic;
      mmio_wr      : in    std_logic;
      mmio_rd      : in    std_logic;
      mmio_addr    : in    std_logic_vector(20 downto 0);
      mmio_wr_data : in    std_logic_vector(31 downto 0);
      mmio_rd_data : out   std_logic_vector(31 downto 0);
      -- switches and LEDs
      sw           : in    std_logic_vector(15 downto 0);
      led          : out   std_logic_vector(15 downto 0);
      -- uart
      rx           : in    std_logic;
      tx           : out   std_logic;
        -- 4 analog input pair
      adc_p        : in    std_logic_vector(3 downto 0);
      adc_n        : in    std_logic_vector(3 downto 0);
      -- pwm
      pwm          : out   std_logic_vector(7 downto 0);
      -- btn
      btn          : in    std_logic_vector(4 downto 0);
      -- 8-digit 7-seg LEDs
      an           : out   std_logic_vector(7 downto 0);
      sseg         : out   std_logic_vector(7 downto 0)
   );
end mmio_sys_sampler;

architecture arch of mmio_sys_sampler is
   signal cs_array       : std_logic_vector(63 downto 0);
   signal reg_addr_array : slot_2d_reg_type;
   signal mem_rd_array   : std_logic_vector(63 downto 0);
   signal mem_wr_array   : std_logic_vector(63 downto 0);
   signal rd_data_array  : slot_2d_data_type;
   signal wr_data_array  : slot_2d_data_type;
   signal adsr_env       : std_logic_vector(15 downto 0);
begin
   --*****************************************************************
   --  MMIO controller instantiation
   --*****************************************************************
   ctrl_unit : entity work.chu_mmio_controller
      port map(
         -- FPro bus interface
         mmio_cs             => mmio_cs,
         mmio_wr             => mmio_wr,
         mmio_rd             => mmio_rd,
         mmio_addr           => mmio_addr,
         mmio_wr_data        => mmio_wr_data,
         mmio_rd_data        => mmio_rd_data,
         -- 64 slot interface
         slot_cs_array       => cs_array,
         slot_reg_addr_array => reg_addr_array,
         slot_mem_rd_array   => mem_rd_array,
         slot_mem_wr_array   => mem_wr_array,
         slot_rd_data_array  => rd_data_array,
         slot_wr_data_array  => wr_data_array
      );
   --*****************************************************************
   -- IO slots instantiations
   --*****************************************************************
   -- slot 0: system timer
   timer_slot0 : entity work.chu_timer
      port map(
         clk     => clk,
         reset   => reset,
         cs      => cs_array(S0_SYS_TIMER),
         read    => mem_rd_array(S0_SYS_TIMER),
         write   => mem_wr_array(S0_SYS_TIMER),
         addr    => reg_addr_array(S0_SYS_TIMER),
         rd_data => rd_data_array(S0_SYS_TIMER),
         wr_data => wr_data_array(S0_SYS_TIMER)
      );
```

```
   -- slot 1: uart1
   uart1_slot1 : entity work.chu_uart
      generic map(FIFO_DEPTH_BIT => 6)
      port map(
         clk     => clk,
         reset   => reset,
         cs      => cs_array(S1_UART1),
         read    => mem_rd_array(S1_UART1),
         write   => mem_wr_array(S1_UART1),
         addr    => reg_addr_array(S1_UART1),
         rd_data => rd_data_array(S1_UART1),
         wr_data => wr_data_array(S1_UART1),
         -- external signals
         tx      => tx,
         rx      => rx
      );
   -- slot 2: GPO for 16 LEDs
   gpo_slot2 : entity work.chu_gpo
      generic map(W => 16)
      port map(
         clk     => clk,
         reset   => reset,
         cs      => cs_array(S2_LED),
         read    => mem_rd_array(S2_LED),
         write   => mem_wr_array(S2_LED),
         addr    => reg_addr_array(S2_LED),
         rd_data => rd_data_array(S2_LED),
         wr_data => wr_data_array(S2_LED),
         -- external signal
         dout    => led
      );
   -- slot 3: input port for 16 slide switches
   gpi_slot3 : entity work.chu_gpi
      generic map(W => 16)
      port map(
         clk     => clk,
         reset   => reset,
         cs      => cs_array(S3_SW),
         read    => mem_rd_array(S3_SW),
         write   => mem_wr_array(S3_SW),
         addr    => reg_addr_array(S3_SW),
         rd_data => rd_data_array(S3_SW),
         wr_data => wr_data_array(S3_SW),
         -- external signal
         din     => sw
      );
   -- slot 4: reserved for user defined
   -- user_slot4 : entity work.
   -- port map(
   --    clk      => clk,
   --    reset    => reset,
   --    cs       => cs_array(S4_USER),
   --    read     => mem_rd_array(S4_USER),
   --    write    => mem_wr_array(S4_USER),
   --    addr     => reg_addr_array(S4_USER),
   --    rd_data  => rd_data_array(S4_USER),
   --    wr_data  => wr_data_array(S4_USER)
   -- );
--   rd_data_array(4) <= (others => '0');
   -- slot 5: xadc
   xadc_slot5 : entity work.chu_xadc_core
      port map(
         clk     => clk,
         reset   => reset,
         cs      => cs_array(S5_XADC),
         read    => mem_rd_array(S5_XADC),
         write   => mem_wr_array(S5_XADC),
         addr    => reg_addr_array(S5_XADC),
         rd_data => rd_data_array(S5_XADC),
         wr_data => wr_data_array(S5_XADC),
         -- external signal
         adc_p   => adc_p,
         adc_n   => adc_n
      );
```

```
--   -- slot 6: pwm
   pwm_slot6 : entity work.chu_io_pwm_core
      generic map(
         W => 8,
         R => 10)
      port map(
         clk     => clk,
         reset   => reset,
         cs      => cs_array(S6_PWM),
         read    => mem_rd_array(S6_PWM),
         write   => mem_wr_array(S6_PWM),
         addr    => reg_addr_array(S6_PWM),
         rd_data => rd_data_array(S6_PWM),
         wr_data => wr_data_array(S6_PWM),
         -- external interface
         pwm_out => pwm
      );
   -- slot 7: push button
   debounce_slot7 : entity work.chu_debounce_core
      generic map(
         W => 5,
         N => 20
      )
      port map(
         clk     => clk,
         reset   => reset,
         cs      => cs_array(S7_BTN),
         read    => mem_rd_array(S7_BTN),
         write   => mem_wr_array(S7_BTN),
         addr    => reg_addr_array(S7_BTN),
         rd_data => rd_data_array(S7_BTN),
         wr_data => wr_data_array(S7_BTN),
         -- external interface
         din     => btn
      );
--   -- slot 8: 7-seg LED
   sseg_led_slot8 : entity work.chu_led_mux_core
      port map(
         clk     => clk,
         reset   => reset,
         cs      => cs_array(S8_SSEG),
         read    => mem_rd_array(S8_SSEG),
         write   => mem_wr_array(S8_SSEG),
         addr    => reg_addr_array(S8_SSEG),
         rd_data => rd_data_array(S8_SSEG),
         wr_data => wr_data_array(S8_SSEG),
         -- external interface
         an      => an,
         sseg    => sseg
   -- assign 0's to all unused slot rd_data signals
   gen_unused_slot : for i in 14 to 63 generate
      rd_data_array(i) <= (others => '0');
   end generate gen_unused_slot;
end arch;
```

El sistema de prueba FPro se puede crear con el subsistema MMIO expandido. Su construcción es similar a la del sistema FPro de vanilla analizado en la sección 5.3.7. El código HDL se muestra en el Listado 5.6.

Listado 5.6. Subsistema de prueba FPro

```
library ieee;
use ieee.std_logic_1164.all;
use work.chu_io_map.all;
entity mcs_top_sampler is
   generic(BRIDGE_BASE : std_logic_vector(31 downto 0) := x"C0000000");
   port(
      clk         : in    std_logic;
      reset_n     : in    std_logic;

      -- switches and LEDs
```

```
      sw          : in    std_logic_vector(15 downto 0);
      led         : out   std_logic_vector(15 downto 0);
      -- uart
      rx          : in    std_logic;
      tx          : out   std_logic;
      -- xadc
      adc_p       : in    std_logic_vector(3 downto 0);
      adc_n       : in    std_logic_vector(3 downto 0);
      -- rgb leds
      rgb_led1    : out   std_logic_vector(2 downto 0);
      rgb_led2    : out   std_logic_vector(2 downto 0);
      -- buttons
      btn         : in    std_logic_vector(4 downto 0);
      -- 4-digit 7-seg LEDs
      an          : out   std_logic_vector(7 downto 0);
      sseg        : out   std_logic_vector(7 downto 0);
   );
end mcs_top_sampler;

architecture arch of mcs_top_sampoler is
   component cpu
      port(
         clk             : in  std_logic;
         reset           : in  std_logic;
         io_addr_strobe  : out std_logic;
         io_read_strobe  : out std_logic;
         io_write_strobe : out std_logic;
         io_address      : out std_logic_vector(31 downto 0);
         io_byte_enable  : out std_logic_vector(3 downto 0);
         io_write_data   : out std_logic_vector(31 downto 0);
         io_read_data    : in  std_logic_vector(31 downto 0);
         io_ready        : in  std_logic
      );
   end component;
   component mmcm_fpro
      port(
         clk_in_100M : in  std_logic;
         clk_100M    : out std_logic;
         clk_25M     : out std_logic;
         clk_40M     : out std_logic;
         clk_67M     : out std_logic;
         reset       : in  std_logic;
         locked      : out std_logic
      );
   end component;

   signal io_addr_strobe  : std_logic;
   signal io_read_strobe  : std_logic;
   signal io_write_strobe : std_logic;
   signal io_byte_enable  : std_logic_vector(3 downto 0);
   signal io_address      : std_logic_vector(31 downto 0);
   signal io_write_data   : std_logic_vector(31 downto 0);
   signal io_read_data    : std_logic_vector(31 downto 0);
   signal io_ready        : std_logic;
   signal mmio_cs         : std_logic;
   signal mmio_wr         : std_logic;
   signal mmio_rd         : std_logic;
   signal mmio_addr       : std_logic_vector(20 downto 0);
   signal mmio_wr_data    : std_logic_vector(31 downto 0);
   signal mmio_rd_data    : std_logic_vector(31 downto 0);
   -- clk/reset related
   signal clk_100M        : std_logic;
   signal clk_25M         : std_logic;
   signal reset_sys       : std_logic;
   signal locked          : std_logic;
   -- pwm
   signal pwm             : std_logic_vector(7 downto 0);
   -- fpro bus
   signal fp_wr             : std_logic;
   signal fp_addr           : std_logic_vector(20 downto 0);
   signal fp_wr_data        : std_logic_vector(31 downto 0);
   signal fp_video_cs       : std_logic;

begin
```

```
   --  rgb leds
   rgb_led2           <= pwm(5 downto 3);
   rgb_led1           <= pwm(2 downto 0);
   -- instantiate clock management unit
   clk_mmcm_unit : mmcm_fpro
      port map(
         -- Clock in ports
         clk_in_100M => clk,
         -- Clock out ports
         clk_100M    => clk_100M,
         clk_25M     => clk_25M,
         clk_40M     => open,
         clk_67M     => open,
         -- Status and control signals
         reset       => '0',
         locked      => locked   );
   -- instantiate microBlaze MCS
   mcs_0 : cpu
      port map(
         clk             => clk_100M,
         reset           => reset_sys,
         io_addr_strobe  => io_addr_strobe,
         io_read_strobe  => io_read_strobe,
         io_write_strobe => io_write_strobe,
         io_byte_enable  => io_byte_enable,
         io_address      => io_address,
         io_write_data   => io_write_data,
         io_read_data    => io_read_data,
         io_ready        => io_ready    );
   -- instantiate MCS IO bus to FPro bus bridge
   bridge_unit : entity work.chu_mcs_bridge
      generic map(BRG_BASE => BRIDGE_BASE)
      port map(
         io_addr_strobe  => io_addr_strobe,
         io_read_strobe  => io_read_strobe,
         io_write_strobe => io_write_strobe,
         io_byte_enable  => io_byte_enable,
         io_address      => io_address,
         io_write_data   => io_write_data,
         io_read_data    => io_read_data,
         io_ready        => io_ready,
         fp_video_cs     => open,
         fp_mmio_cs      => mmio_cs,
         fp_wr           => mmio_wr,
         fp_rd           => mmio_rd,
         fp_addr         => mmio_addr,
         fp_wr_data      => mmio_wr_data,
         fp_rd_data      => mmio_rd_data  );
   -- instantiate sampler MMIO subsystem
   mmio_sys_unit : entity work.mmio_sys_sampler
      port map(
         clk          => clk_100M,
         reset        => reset_sys,
         mmio_cs      => mmio_cs,
         mmio_wr      => mmio_wr,
         mmio_rd      => mmio_rd,
         mmio_addr    => mmio_addr,
         mmio_wr_data => mmio_wr_data,
         mmio_rd_data => mmio_rd_data,
         sw           => sw,
         led          => led,
         rx           => rx,
         tx           => tx,
         adc_p        => adc_p,
         adc_n        => adc_n,
         pwm          => pwm,
         btn          => btn,
         an           => an,
         sseg         => sseg );
end arch;
```

5.5.4.4 Programa de verificación de software

El programa de prueba para el sistema de muestra FPro se visualiza en el Listado 5.7. Se expande el programa de prueba del sistema vanilla FPro descrito en la sección 5.2.8.4 para los bloques adicionales. Además de las funciones de prueba, el programa principal también incluye una función auxiliar, `show_test_id()`, que parpadea el patrón binario en LEDs discretos durante unos segundos para inducir el número de la prueba actual. Las partes principales de estas funciones de prueba se pueden encontrar en los capítulos respectivos.

Listado 5.7. Programa de prueba FPro (main_sampler_test.cpp)

```
#include "chu_init.h"
#include "gpio_cores.h"
#include "xadc_core.h"
#include "sseg_core.h"
#include "spi_core.h"
#include "i2c_core.h"
#include "ps2_core.h"
#include "ddfs_core.h"
#include "adsr_core.h"

void timer_check(GpoCore *led_p) { ... }
void led_check(GpoCore *led_p, int n) { ... }
void sw_check(GpoCore *led_p, GpiCore *sw_p) { ... }
void uart_check() { ... }
void adc_check(XadcCore *adc_p, GpoCore *led_p) { ... }
void pwm_3color_led_check(PwmCore *pwm_p) { ... }
void debounce_check(DebounceCore *db_p, GpoCore *led_p) { ...}
void sseg_check(SsegCore *sseg_p) { ... }
void gsensor_check(SpiCore *spi_p, GpoCore *led_p) { ... }
void adt7420_check(I2cCore *adt7420_p, GpoCore *led_p) { ... }
void ps2_check(Ps2Core *ps2_p) { ... }
void ddfs_check(DdfsCore *ddfs_p, GpoCore *led_p) { ... }
void adsr_check(AdsrCore *adsr_p, GpoCore *led_p, GpiCore *sw_p) { ... }

void show_test_id(int n, GpoCore *led_p) {
   int i, ptn;

   ptn = n; //1 << n;
   for (i = 0; i < 20; i++) {
      led_p->write(ptn);
      sleep_ms(30);
      led_p->write(0);
      sleep_ms(30);
   }
}

GpoCore led(get_slot_addr(BRIDGE_BASE, S2_LED));
GpiCore sw(get_slot_addr(BRIDGE_BASE, S3_SW));
XadcCore adc(get_slot_addr(BRIDGE_BASE, S5_XDAC));
PwmCore pwm(get_slot_addr(BRIDGE_BASE, S6_PWM));
DebounceCore btn(get_slot_addr(BRIDGE_BASE, S7_BTN));
SsegCore sseg(get_slot_addr(BRIDGE_BASE, S8_SSEG));
SpiCore spi(get_slot_addr(BRIDGE_BASE, S9_SPI));
I2cCore adt7420(get_slot_addr(BRIDGE_BASE, S10_I2C));
Ps2Core ps2(get_slot_addr(BRIDGE_BASE, S11_PS2));
DdfsCore ddfs(get_slot_addr(BRIDGE_BASE, S12_DDFS));
AdsrCore adsr(get_slot_addr(BRIDGE_BASE, S13_ADSR), &ddfs);

int main() {
   timer_check(&led);
   while (1) {
      show_test_id(1, &led);
      led_check(&led, 16);
      sw_check(&led, &sw);
      show_test_id(3, &led);
```

```
      uart_check();
      debug("main - switch value / up time : ", sw.read(), now_ms());
      show_test_id(5, &led);
      adc_check(&adc, &led);
      show_test_id(6, &led);
      pwm_3color_led_check(&pwm);
      show_test_id(7, &led);
      debounce_check(&btn, &led);
      show_test_id(8, &led);
      sseg_check(&sseg);
      show_test_id(9, &led);
      gsensor_check(&spi, &led);
      show_test_id(10, &led);
      adt7420_check(&adt7420, &led);
      show_test_id(11, &led);
      ps2_check(&ps2);
      show_test_id(12, &led);
      ddfs_check(&ddfs, &led);
      show_test_id(13, &led);
      adsr_check(&adsr, &led, &sw);
   } //while
} //main
```

5.6 BLOQUE MODULACIÓN POR ANCHURA DE PULSO

PWM (modulación por anchura de pulso) es un esquema para codificar niveles de señales analógicas, como un motor o una luz. En este capítulo, se desarrolla un bloque MMIO para generar múltiples canales PWM, cuya composición de E/S se muestra a continuación:.

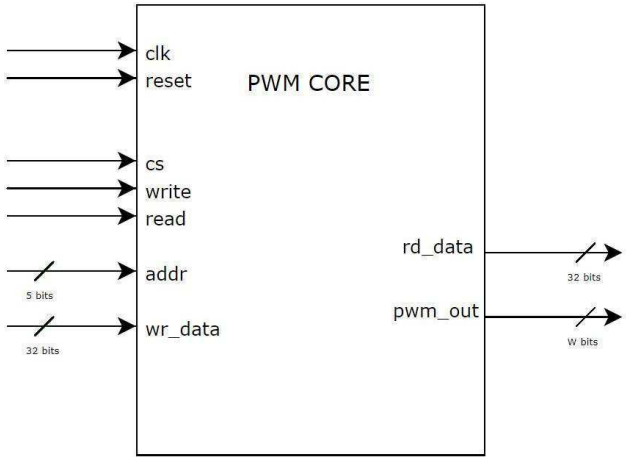


Ilustración 32: *Composición de E/S bloque PWM*

5.6.1 INTRODUCCIÓN

5.6.1.1 PWM como salida analógica

Una onda cuadrada oscila entre cero y uno donde el ciclo de trabajo es la proporción de "intervalo activo (un 1 lógico)" dentro de un período. Se considera una onda cuadrada con un período de t_{PWM} y un intervalo de t_{ON}, como se muestra en la Ilustración 33 (a).

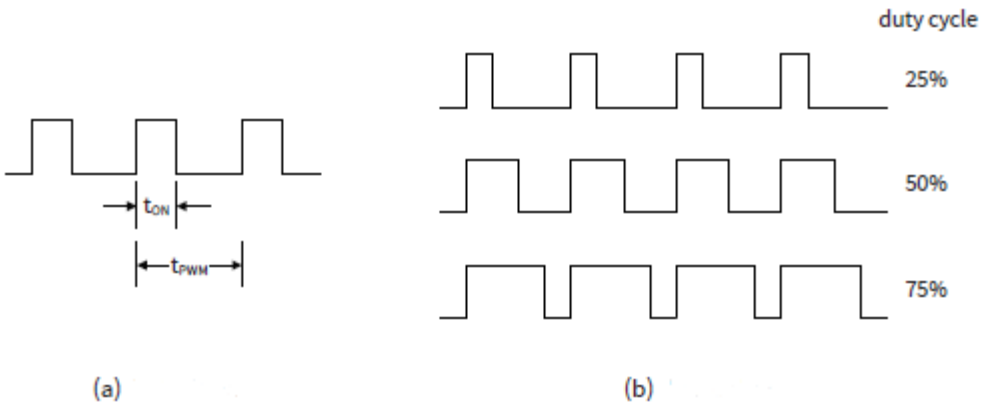


Ilustración 33: *Definición (a) y ejemplos (b) de ciclo de trabajo.*

Las formas de onda de diferentes ciclos de trabajo se muestran en Ilustración 33 (b).

El PWM es un esquema para ajustar el ciclo de trabajo de una onda cuya señal puede considerarse como una forma específica de señal analógica. Se considera que los voltajes de lógica 0 y lógica 1 sean 0 y Vcc y el ciclo de trabajo sea *d*. El nivel de voltaje promedio es *d* * Vcc (d * Vcc + (1 - *d*) * 0). Por lo tanto, mientras que los valores de voltaje de salida instantáneos son "digitales" (0 o Vcc), el voltaje promedio es "analógico" y puede incrementarse gradualmente de 0 a Vcc. El esquema PWM se puede considerar como un método especial de conversión de digital a analógico que incorpora el valor analógico en su ciclo de trabajo.

5.6.1.2 Características principales

Las dos características importantes de PWM son la frecuencia de conmutación y la resolución. La frecuencia de conmutación es la frecuencia de onda cuadrada, que es $\frac{1}{t_{PWM}}$ que se observa en la Ilustración 33. La frecuencia de conmutación exacta depende de la naturaleza de la aplicación. Por ejemplo, podemos usar PWM para controlar el brillo de un LED. Si la frecuencia de conmutación es demasiado baja (10 Hz), puede parecer que el LED está parpadeando. Por otro lado, si la frecuencia de conmutación es demasiado alta (1 MHz), es posible que el LED no tenga tiempo suficiente para encenderse o apagarse por completo. Esta frecuencia puede tratar de unos pocos KHz a decenas de KHz para una unidad de motor y en decenas o cientos de KHz en amplificadores de audio y fuentes de alimentación.

La resolución describe la "granularidad" (el paso incremental mínimo) del ciclo de trabajo. Se especifica en términos de número de bits. Para una resolución de R bits, hay 2^R niveles en el ciclo de trabajo y el paso incremental mínimo es $\frac{1}{2^R}$. Por ejemplo, la granularidad de un PWM de 8 bits es $\left(\frac{1}{2^8}\right)$ y sus ciclos de trabajo son... $\frac{0}{256}$, $\frac{1}{256}$, $\frac{2}{256}$, $\frac{3}{256}$, … , $\frac{255}{256}$, $\frac{256}{256}$.

5.6.2 DISEÑO PWM

5.6.2.1 Diseño básico

El circuito básico de PWM consiste en un contador binario y un comparador, como se muestra en la Ilustración 34. La entrada del ciclo de trabajo deseada, `duty`, establece el umbral para el comparador y la salida del comparador se valida cuando el valor del contador está por debajo del umbral. Por ejemplo, suponemos que la resolución del PWM es de ocho bits. El contador cuenta de 0 a 255 y se resetea. Si el valor de la señal `duty` es *d*, la salida del comparador se activa cuando el valor del contador es 0, 1, 2,... *d* - 1. Por lo tanto, se confirma *d*

de 256 ciclos de reloj, y ciclo de trabajo es d / 256. Se agrega un regiostro D a la salida del comparador para filtrar posibles fallas.

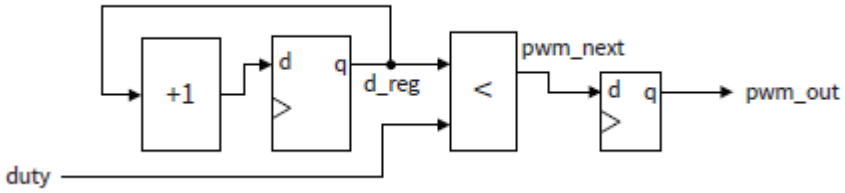


Ilustración 34: *Diagrama de bloques del circuito básico de PWM.*

El código HDL del circuito PWM básico sigue el diagrama de la Figura 6.2 y se muestra en el Listado 6.1.

Listado 6.1. Circuito básico PWM

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity pwm_basic is
   generic(
      R : integer := 8   -- # bits of PWM resolution (i.e., 2^R levels)
   );
   port(
      clk     : in  std_logic;
      reset   : in  std_logic;
      duty    : in  std_logic_vector(R - 1 downto 0);
      pwm_out : out std_logic
   );
end pwm_basic;

architecture arch of pwm_basic is
   signal d_reg, d_next     : unsigned(R - 1 downto 0);
   signal pwm_reg, pwm_next : std_logic;
begin
   process(clk, reset)
   begin
      if reset = '1' then
         d_reg <= (others => '0');
         pwm_reg <=  '0';
      elsif (clk'event and clk = '1') then
         d_reg <= d_next;
         pwm_reg <=  pwm_next;
     end if;
   end process;
   -- duty cycle counter
   d_next   <= d_reg + 1;
   -- comparison circuit
   pwm_next <= '1' when d_reg < unsigned(duty) else '0';
   pwm_out  <= pwm_reg;
end arch;
```

Un genérico, R, representa el número de bits en el contador y, por lo tanto, define la resolución de la PWM.

5.6.2.2 Diseño mejorado

El circuito básico de PWM sufre dos problemas. Primero, no puede controlar la frecuencia de conmutación PWM. Deje que la velocidad de reloj del sistema sea fsys y la resolución PWM sea R bits. Hay 2^R ciclos de reloj en un período PWM (t_{PWM} en la Figura 6.1).

Así, la frecuencia de conmutación de la onda PWM se convierte en:

$$f_{pwm} = \frac{f_{sys}}{2^R}$$

Para un PWM con un reloj de sistema de 100 MHz y 8 bits, f_{pwm} es de aproximadamente 390 KHz, lo que es demasiado alto para muchas aplicaciones. Se puede usar un divisor de frecuencia para producir una señal de reloj para impulsar el contador de `duty`. Si la frecuencia PWM deseada es f_p, el valor del divisor debe satisfacer

$$f_p = \frac{\frac{f_{sys}}{dvsr}}{2^R}$$

Por lo tanto, el valor del divisor debe ser

$$dvsr = \frac{f_{sys}}{2^R * f_p}$$

El segundo problema es el ciclo de trabajo al 100%. Consideramos un PWM de 8 bits en el que el valor de la señal `duty` es *d*. La salida del comparador se afirma cuando el valor del contador es 0, 1, 2, ..., *d* - 1 y, por lo tanto, su ciclo de trabajo es $\frac{d}{256}$. Sin embargo, como la señal `duty` tiene una anchura de 8 bits y el valor máximo es de 255, el ciclo de trabajo máximo que se puede alcanzar es de $\frac{255}{256}$, que no es del 100% $\left(\frac{256}{256}\right)$. Una prueba más detallada muestra que *d* contiene 257 valores (0, 1, 2, ..., 254, 255, 256). La señal `duty` debe extenderse en un bit adicional para acomodar el valor adicional, el rango de entrada debe extenderse de [0x00, 0xff] a [0x000, 0x100].

El diseño mejorado de PWM incorpora el contador del divisor de frecuencia y el circuito de comparación que extiende el ancho del puerto `duty`. Su código HDL se muestra en el Listado 6.2. Para lograr flexibilidad, el valor del divisor de frecuencia se especifica mediante una señal externa, `dvsr`.

Listado 6.2. Circuito PWM mejorado

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity pwm_enhanced is
   generic(
      R : integer := 10
   );
   port(
      clk     : in  std_logic;
      reset   : in  std_logic;
      dvsr    : in  std_logic_vector(31 downto 0);
      duty    : in std_logic_vector(R downto 0);
      pwm_out : out std_logic
   );
end pwm_enhanced;

architecture arch of pwm_enhanced is
   signal q_reg, q_next     : unsigned(31 downto 0);
   signal d_reg, d_next     : unsigned(R - 1 downto 0);
   signal d_ext             : unsigned(R downto 0);
   signal pwm_reg, pwm_next : std_logic;
   signal tick              : std_logic;
begin
   process(clk, reset)
   begin
      if reset = '1' then
         q_reg   <= (others => '0');
         d_reg   <= (others => '0');
         pwm_reg <=  '0';
      elsif (clk'event and clk = '1') then
         q_reg   <= q_next;
         d_reg   <= d_next;
         pwm_reg <= pwm_next;
      end if;
   end process;
   -- "prescaler" counter
   q_next <=  (others => '0') when q_reg=unsigned(dvsr) else q_reg + 1;
   tick   <= '1' when q_reg = 0 else '0';
   -- duty cycle counter
   d_next <= d_reg + 1 when tick = '1' else d_reg;
   d_ext  <= '0' & d_reg;
   -- comparison circuit
   pwm_next <= '1' when d_ext < unsigned(duty) else '0';
   pwm_out  <= pwm_reg;
end arch;
```

5.6.3 DESARROLLO DEL PWM CORE

El GPO core discutido en la sección 5.3.3.3 genera múltiples salidas. Cada salida asume un valor de lógica 0 (0 V) o lógica 1 (Vcc) y, por lo tanto, es de naturaleza "digital". El circuito PWM genera una señal con un ciclo de trabajo ajustable, que puede tratarse como una salida "analógica" de un bit. Un bloque PWM contiene múltiples circuitos PWM y puede considerarse como un bloque que genera múltiples salidas "analógicas". Para simplificar, asumimos que las salidas PWM se utilizan para controlar dispositivos similares (LED) y, por lo tanto, se aplica la misma frecuencia de conmutación a todos los circuitos PWM en el núcleo.

5.6.3.1 Mapa de registro

El procesador interactúa con los circuitos PWM de la siguiente manera:

- Especifica (escribe) la frecuencia de conmutación a través de la señal `dvsr`.
- Establece (escribe) el valor del ciclo de trabajo para cada salida.

El mapa de registro del PWM core es algo diferente de otros bloques. Debido a que la configuración del ciclo de trabajo requiere múltiples bits (la resolución del circuito PWM), cada salida PWM necesita un registro. Por lo tanto, la cantidad de registros en el bloque no es fija sino que depende de la cantidad de salidas PWM.

Supongamos que un núcleo PWM tiene *w + 1* salidas. Los offsets de dirección y campos de los registros son:

- offset 0x00 (registro divisor)
    - bits 31 a 0: valor del contador de divisor de frecuencia
- offset 0x10 (registro de ciclo de trabajo para PWM bit 0)
    - bits R a 0: valor del ciclo de trabajo (donde R es la resolución del circuito PWM)
- offset 0x11 (registro de ciclo de trabajo para PWM bit 1)
    - bits R a 0: valor del ciclo de trabajo
- offset 0x12 (registro de ciclo de trabajo para PWM bit 2)
    - bits R a 0: valor del ciclo de trabajo

- offset 0x1*w* (registro de ciclo de trabajo para PWM bit *w*)
    - bits R a 0: valor del ciclo de trabajo

El número máximo de salidas PWM es 16.

5.6.3.2 Circuito de envoltura PWM

Para cumplir con la especificación del slot o ranura, podemos construir un circuito de envoltura que contenga registros y un circuito de decodificación para almacenar los valores del divisor y del ciclo de trabajo. El código HDL del PWM core se muestra en el Listado 6.3.

Listado 6.3. Núcelo PWM

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity chu_io_pwm_core is
   generic(
      W : integer := 8;    -- width (# bits) of output port
      R : integer := 8     -- # bits of PWM resolution (2^R levels)
   );
   port(
      clk     : in  std_logic;
      reset   : in  std_logic;
      -- slot interface
      cs      : in  std_logic;
      write   : in  std_logic;
      read    : in  std_logic;
      addr    : in  std_logic_vector(4 downto 0);
      rd_data : out std_logic_vector(31 downto 0);
      wr_data : in  std_logic_vector(31 downto 0);
      -- external signals
      pwm_out : out std_logic_vector(W - 1 downto 0)
   );
end chu_io_pwm_core;

architecture arch of chu_io_pwm_core is
   type reg_file_type is array (W - 1 downto 0) of
        std_logic_vector(R downto 0);
   signal duty_2d_reg    : reg_file_type;
   signal wr_en, dvsr_en : std_logic;
   signal duty_array_en  : std_logic;
   signal q_reg, q_next  : unsigned(31 downto 0);
   signal d_reg, d_next  : unsigned(R - 1 downto 0);
   signal d_ext          : unsigned(R downto 0);
   signal pwm_next       : std_logic_vector(W - 1 downto 0);
   signal pwm_reg        : std_logic_vector(W - 1 downto 0);
   signal tick           : std_logic;
   signal dvsr_reg       : std_logic_vector(31 downto 0);
begin
   --*****************************************************************
   -- wrapping circuit
   --*****************************************************************
   --  decoding logic
   wr_en         <= '1' when write = '1' and cs = '1' else '0';
   duty_array_en <= '1' when wr_en = '1' and addr(4) = '1' else '0';
   dvsr_en       <= '1' when wr_en = '1' and addr = "00000" else '0';
   -- register for divisor
   process(clk, reset)
   begin
      if (reset = '1') then
         dvsr_reg <= (others => '0');
```

```
      elsif (clk'event and clk = '1') then
         if dvsr_en = '1' then
            dvsr_reg <= wr_data;
         end if;
      end if;
   end process;
   -- register file for duty cycles
   process(clk, reset)
   begin
      if (reset = '1') then
         duty_2d_reg <= (others => (others => '0'));
      elsif (clk'event and clk = '1') then
         if duty_array_en = '1' then
           duty_2d_reg(to_integer(unsigned(addr(3 downto 0))))<=wr_data;
         end if;
      end if;
   end process;
   --*****************************************************************
   -- multi-bit PWM
   --*****************************************************************
   process(clk, reset)
   begin
      if reset = '1' then
         q_reg   <= (others => '0');
         d_reg   <= (others => '0');
         pwm_reg <= (others => '0');
      elsif (clk'event and clk = '1') then
         q_reg   <= q_next;
         d_reg   <= d_next;
         pwm_reg <= pwm_next;
      end if;
   end process;
   -- "prescale" counter
   q_next <= (others=>'0') when q_reg=unsigned(dvsr_reg) else q_reg + 1;
   tick   <= '1' when q_reg = 0 else '0';
   -- duty cycle counter
   d_next <= d_reg + 1 when tick = '1' else d_reg;
   d_ext  <= '0' & d_reg;
   -- comparison circuit
   gen_comp_cell : for i in 0 to W - 1 generate
      pwm_next(i) <= '1' when d_ext<unsigned(duty_2d_reg(i)) else '0';
   end generate;
   pwm_out <= pwm_reg;
   -- read data not used
   rd_data <= (others => '0');
end arch;
```

El código se compone de dos partes principales. La primera parte es el circuito de envoltura que contiene un circuito de decodificación, un registro para el divisor y un archivo de registro bidimensional para los ciclos de trabajo. El tamaño del archivo de registro está especificado por la variable *W* genérica y, por lo tanto, puede ajustarse para adaptarse al número deseado de registros de ciclo de trabajo.

La segunda parte es el circuito PWM de múltiples bits. El código es similar al del Listado 6.2, excepto que se obtienen múltiples circuitos de comparación de la declaración generada. Hay que tener en cuenta que todos los bits PWM comparten el contador del divisor de frecuencia y el contador de `duty`.

5.6.4 CONTROLADOR PWM

El controlador PWM consta de un conjunto de rutinas para configurar la frecuencia de conmutación PWM y el ciclo de trabajo de los canales individuales PWM. Como la señal PWM se considera una forma de salida "analógica", la definición de clase y la implementación se incluyen en los archivos `gpio_core.h` y `gpio_core.cpp`.

5.6.4.1 Definición de clase

La definición de clase del bloque de PWM se muestra en el Listado 6.4.

Listado 6.4. Definición de clase `PwmCore` (`gpio_core.h`)

```
   enum {
      DVSR_REG = 0,         /**< pwm divisor register */
      DUTY_REG_BASE = 0x10  /**< channel 0 duty cycle register */
   };
   enum {
      RESOLUTION_BITS = 10,
      MAX = 1 << RESOLUTION_BITS
   };
public:
   PwmCore(uint32_t core_base_addr);
   ~PwmCore();

   void set_freq(int freq);
   void set_duty(int duty, int channel);
   void set_duty(double f, int channel);

private:
   uint32_t base_addr;
};
```

La definición de enumeración utiliza nombres simbólicos para los offsets de registro de divisor y el offset base de los registros de ciclo de trabajo. La segunda definición de enumeración especifica la resolución PWM, que se define en el código HDL cuando se crea una instancia del núcleo PWM, y el valor del ciclo de trabajo máximo, que es $2^{RESOLUTION\_BITS}$ que se establece en 10 bits en nuestro bloque instanciado.

5.6.4.2 Implementación de clase

La implementación de clase del constructor y los métodos se muestran en el Listado 6.5.

Listado 6.5. Implementación de clase `PwmCore` (`gpio_core.cpp`)

```
   PwmCore::PwmCore(uint32_t core_base_addr) {
   base_addr = core_base_addr;
   set_freq(1000);
}

PwmCore::~PwmCore() {
}

void PwmCore::set_freq(int freq) {
   uint32_t dvsr;
   dvsr = (uint32_t) SYS_CLK_FREQ * 1000000 / MAX / freq;
   io_write(base_addr, DVSR_REG, dvsr);
}

void PwmCore::set_duty(int duty, int channel) {
   uint32_t d;

   if (duty > MAX) {
      d = MAX;
   } else {
      d = duty;
   }
   io_write(base_addr, DUTY_REG_BASE + channel, d);
}

void PwmCore::set_duty(double f, int channel) {
   int duty;
   duty = (int) (f * MAX);
   debug("set_duty_f: ", f, duty);
   set_duty(duty, channel);
}
```

El constructor `PwmCore()` guarda la dirección base y establece la frecuencia de conmutación PWM predeterminada en 1000 Hz. El método `set_freq()` calcula el valor del divisor a partir de la frecuencia de conmutación deseada y escribe el valor en el registro del divisor. Los métodos `set_duty()` escriben el valor del ciclo de trabajo en el registro designado. El ciclo de trabajo se puede expresar como un número entero (como con el parámetro de trabajo) con un rango entre 0 y `MAX`, y los datos sin procesar se utilizarán directamente. Alternativamente, el ciclo de trabajo puede expresarse como un número entero real (como con el parámetro *f*) con un rango entre 0.0 y 1.0, que representa un ciclo de trabajo de 0 y 100%. El último método convierte el número real en entero y escribe el entero en el registro designado. Se prefiere este método ya que el valor del ciclo de trabajo es independiente de la resolución.

5.6.5 PRUEBAS

El sistema de prueba FPro crea una instancia central de PWM de ocho bits. Se usan seis canales de salida para controlar dos LED tricolor en la placa Nexys 4 DDR y dos canales están conectados a las dos conexiones de un puerto PMOD.

La rutina de prueba demuestra el funcionamiento básico de PWM al aumentar gradualmente el brillo de los LED tricolor incrementando los valores del ciclo de trabajo. El código se muestra en el Listado 6.6.

Listado 6.6. Programa de prueba PWM (`main_sampler_test.cpp`)

```
   void pwm_3color_led_check(PwmCore *pwm_p) {
   int i, n;
   double bright, duty;
   const double P20 = 1.2589;  // P20=100^(1/20); i.e., P20^20=100

   pwm_p->set_freq(50);
   for (n = 0; n < 3; n++) {
      bright = 1.0;
      for (i = 0; i < 20; i++) {
         bright = bright * P20;
         duty = bright / 100.0;
         pwm_p->set_duty(duty, n);
         pwm_p->set_duty(duty, n + 3);
         sleep_ms(100);
      }
      sleep_ms(300);
      pwm_p->set_duty(0.0, n);
      pwm_p->set_duty(0.0, n + 3);
   }
}
```

5.7 BLOQUE ANTIRREBOTE Y BLOQUE LED-MUX

En este apartado, se demuestra cómo convertir un circuito antirrebote y un circuito de multiplexación en MMIO cores e incorporarlos en un sistema FPro.

5.7.1 BLOQUE ANTIRREBOTE

Un circuito antirrebote filtra los rebotes de contacto transitorios de un interruptor mecánico. Es como un bloque de entrada de propósito general pero con transiciones de entrada "limpiadas". La estructura de un módulo antirrebote en cuanto al conjunto de E/S es la siguiente:

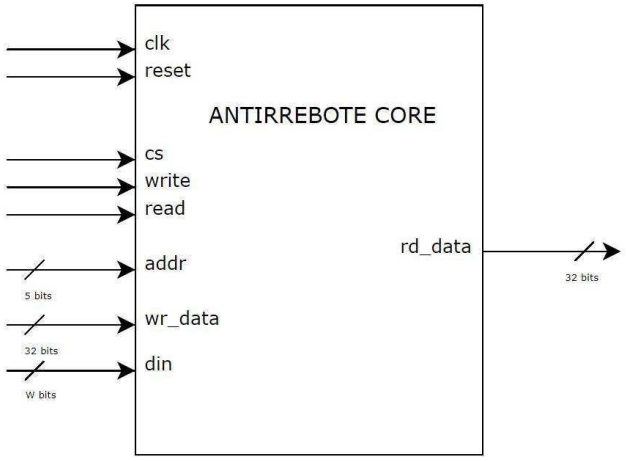


Ilustración 35: *Composición de E/S bloque Antirrebote*

5.7.1.1 Circuito antirrebote multi-bit

Un circuito antirrebote usa un temporizador separado con una máquina de estados finitos (*FSM (finite state machine)*). Para facilitar el desarrollo del bloque, el circuito debe expandirse para admitir múltiples bits de entrada. Una forma sencilla de lograr esto es replicar el circuito original de un bit varias veces con una declaración de generación. Sin embargo, este enfoque no es muy eficiente. Cuando se sintetiza el circuito, el temporizador, que cuenta para el intervalo de 10 ms, infiere un gran incremento. Replicarlo varias veces es un desperdicio de recursos de hardware.

Para el diseño de FSM, un temporizador está separado del circuito de FSM y, por lo tanto, puede compartirse según sea necesario. Por lo tanto, un circuito antirrebote de múltiples bits solo necesita replicar el FSM. Para facilitar el desarrollo, el código original se divide en un módulo FSM y un módulo de temporizador, como se muestra en los Listados 7.1 y 7.2.

Listado 7.1. FSM Antirrebote

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity debounce_fsm is
   port(
      clk, reset : in  std_logic;
      sw         : in  std_logic;
      m_tick     : in  std_logic;
      db         : out std_logic
   );
end debounce_fsm;

architecture arch of debounce_fsm is
   type eg_state_type is
     (zero, wait1_1, wait1_2, wait1_3, one, wait0_1, wait0_2, wait0_3);
   signal state_reg, state_next : eg_state_type;
begin
   -- state register
   process(clk, reset)
   begin
      if (reset = '1') then
         state_reg <= zero;
      elsif (clk'event and clk = '1') then
         state_reg <= state_next;
      end if;
   end process;
   -- next-state/output logic
   process(state_reg, sw, m_tick)
   begin
      state_next <= state_reg;
      db         <= '0';
      case state_reg is
         when zero =>
            if sw = '1' then
               state_next <= wait1_1;
            end if;
         when wait1_1 =>
            if sw = '0' then
               state_next <= zero;
            else
               if m_tick = '1' then
                  state_next <= wait1_2;
               end if;
            end if;
         when wait1_2 =>
            if sw = '0' then
               state_next <= zero;
            else
               if m_tick = '1' then
                  state_next <= wait1_3;
               end if;
            end if;
         when wait1_3 =>
            if sw = '0' then
               state_next <= zero;
            else
               if m_tick = '1' then
                  state_next <= one;
               end if;
            end if;
         when one =>
            db <= '1';
            if sw = '0' then
               state_next <= wait0_1;
            end if;
         when wait0_1 =>
            db <= '1';
            if sw = '1' then
               state_next <= one;
            else
               if m_tick = '1' then
```

```
                state_next <= wait0_2;
             end if;
          end if;
       when wait0_2 =>
          db <= '1';
          if sw = '1' then
             state_next <= one;
          else
             if m_tick = '1' then
                state_next <= wait0_3;
             end if;
          end if;
       when wait0_3 =>
          db <= '1';
          if sw = '1' then
             state_next <= one;
          else
             if m_tick = '1' then
                state_next <= zero;
             end if;
          end if;
      end case;
   end process;
end arch;
```

Listado 7.2. Temporizador del bloque Antirrebote

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity debounce_counter is
   generic(N : integer := 20);  -- 2^N * 10ns = 10ms tick
   port(
      clk, reset : in  std_logic;
      m_tick     : out std_logic
   );
end debounce_counter;

architecture arch of debounce_counter is
   signal q_reg, q_next : unsigned(N - 1 downto 0);
begin
   process(clk, reset)
   begin
      if (reset = '1') then
         q_reg <= (others => '0');
      elsif (clk'event and clk = '1') then
         q_reg <= q_next;
      end if;
   end process;
   q_next <= q_reg + 1;
   --output tick
   m_tick <= '1' when q_reg = 0 else '0';
end arch;
```

5.7.1.2 Registrar mapa y el circuito de envoltura de ranura.

El bloque antirrebote es como un bloque de entrada de propósito general que filtra los rebotes de transitorios no deseados. Para que sea más versátil, el bloque proporciona señales de entrada sin tratar y sin rebote. El mapa de registro se puede definir en consecuencia. Hay dos registros de lectura. Sus desplazamientos y campos de dirección son:

- offset 0 (registro de datos de entrada sin procesar)
    - bits *W-1* a 0: entradas sin tratar de *W* bits, donde *W* es el valor genérico de VHDL.
- offset 1 (registro de datos de entrada antirrebote)
    - Bits *W*-1 a 0: entradas antirrebote de *W* bits.

Sobre la base del mapa de registro, podemos derivar un circuito de envoltura que cumpla con la especificación de la ranura y crear el bloque de antirrebote. El código HDL se muestra en el Listado 7.3. Consiste en un registro de datos de entrada para entradas sin procesar, una instancia de un temporizador, varias instancias de los FSM de antirrebote y un circuito de multiplexación de lectura. Los múltiples FSM se obtienen mediante una declaración de generación y, por lo tanto, comparten el mismo temporizador.

Listado 7.3. Bloque Antirrebote

```
library ieee;
use ieee.std_logic_1164.all;
entity chu_debounce_core is
   generic(
      W : integer := 8;    -- width of input port
      N : integer := 20    -- # bit for 10-ms tick 2^N * clk period
   );
   port(clk     : in  std_logic;
        reset   : in  std_logic;
        -- io bridge interface
        cs      : in  std_logic;
        write   : in  std_logic;
        read    : in  std_logic;
        addr    : in  std_logic_vector(4 downto 0);
        rd_data : out std_logic_vector(31 downto 0);
        wr_data : in  std_logic_vector(31 downto 0);
        -- external signal
        din     : in  std_logic_vector(W-1 downto 0));
end chu_debounce_core;
architecture arch of chu_debounce_core is
   signal rd_data_reg : std_logic_vector(W-1 downto 0);
   signal m_tick      : std_logic;
   signal db_out      : std_logic_vector(W-1 downto 0);
begin
   --*****************************************************************
   -- input register
   --*****************************************************************
   process(clk, reset)
   begin
      if reset = '1' then
         rd_data_reg <= (others => '0');
      elsif (clk'event and clk = '1') then
         rd_data_reg <= din;
      end if;
   end process;
   --*****************************************************************
   -- instantiate one counter and W debouncing FSMs
   --*****************************************************************
   db_counter_unit : entity work.debounce_counter
      generic map(N => N)
      port map(
         clk    => clk,
         reset  => reset,
         m_tick => m_tick  );
   gen_fsm_cell : for i in 0 to W-1 generate
```

```
      db_fsm_unit : entity work.debounce_fsm
         port map(
            clk    => clk,
            reset  => reset,
            sw     => din(i),
            m_tick => m_tick,
            db     => db_out(i)
         );
   end generate;
   --*****************************************************************
   -- read multiplexing
   --*****************************************************************
   rd_data(W-1 downto 0) <= rd_data_reg when addr(0)='0' else db_out;
   rd_data(31 downto W)  <= (others => '0');
end arch;
```

5.7.1.3 Controlador antirrebote

El controlador del antirrebote core contiene métodos para leer la entrada no procesada y la entrada antirrebote. La definición de clase y la implementación se muestran en los listados 7.4 y 7.5. Debido a que una entrada antirrebote es una forma específica de entrada, la definición y la implementación de la clase se incluyen en los archivos `gpio_core.h` y `gpio_core.cpp`.

Listado 7.4. Definición de clase `DebounceCore` (`gpio_core.h`)

```
class DebounceCore {
   /* register map */
   enum {
      NORMAL_DATA_REG = 0, /**< un-treated input data register */
      DB_DATA_REG = 1      /**< debounced input data register */
   };
public:
   DebounceCore(uint32_t core_base_addr);
   ~DebounceCore();                  // not used
   uint32_t read();
   int read(int bit_pos);
   uint32_t read_db();
   int read_db(int bit_pos);
private:
   uint32_t base_addr;
end arch;
```

Listado 7.5. Implementación de clase `DebounceCore` (`gpio_core.cpp`)

```
DebounceCore::DebounceCore(uint32_t core_base_addr) {
   base_addr = core_base_addr;
}
DebounceCore::~DebounceCore() {
}

uint32_t DebounceCore::read() {
   return (io_read(base_addr, NORMAL_DATA_REG));
}

int DebounceCore::read(int bit_pos) {
   uint32_t rd_data = io_read(base_addr, NORMAL_DATA_REG);
   return ((int) bit_read(rd_data, bit_pos));
}

uint32_t DebounceCore::read_db() {
   return (io_read(base_addr, DB_DATA_REG));
}

int DebounceCore::read_db(int bit_pos) {
   uint32_t rd_data = io_read(base_addr, DB_DATA_REG);
   return ((int) bit_read(rd_data, bit_pos));
}
```

5.7.1.4 Prueba

El sistema de prueba FPro crea una instancia del bloque antirrebote de cinco bits y conecta las entradas a los cinco botones en la placa Nexys 4 DDR.

La rutina de prueba cuenta las transiciones en cinco botones que usan tanto entradas sin procesar como entradas antirrebote y muestra los resultados en los LED discretos. El código se muestra en el Listado 7.6. El número de rebotes depende del interruptor individual. Algunos interruptores pueden exhibir muy pocos rebotes.

Listado 7.6. Programa de prueba Antirrebote (`main_sampler_test.cpp`)

```
void debounce_check(DebounceCore *db_p, GpoCore *led_p) {
   long start_time;
   int btn_old, db_old, btn_new, db_new;
   int b = 0;
   int d = 0;
   uint32_t ptn;

   start_time = now_ms();
   btn_old = db_p->read();
   db_old = db_p->read_db();
   do {
      btn_new = db_p->read();
      db_new = db_p->read_db();
      if (btn_old != btn_new) {
         b = b + 1;
         btn_old = btn_new;
      }
      if (db_old != db_new) {
         d = d + 1;
         db_old = db_new;
      }
      ptn = d & 0x0000000f;
      ptn = ptn | (b & 0x0000000f) << 4;
      led_p->write(ptn);
   } while ((now_ms() - start_time) < 5000);
}
```

5.7.2 BLOQUE LED-MUX

Esta compuesto por un circuito de multiplexación LED que realiza una multiplexación de tiempo para reducir el número de pines de E/S. Se presenta el circuito para acomodar el display de ocho dígitos de 7 segmentos de la placa Nexys 4 DDR y se convierte en un MMIO core para el sistema FPro. En cuanto al módulo LED-Mux, presenta una organización de entradas y salidas como se muestra en la siguiente imagen:

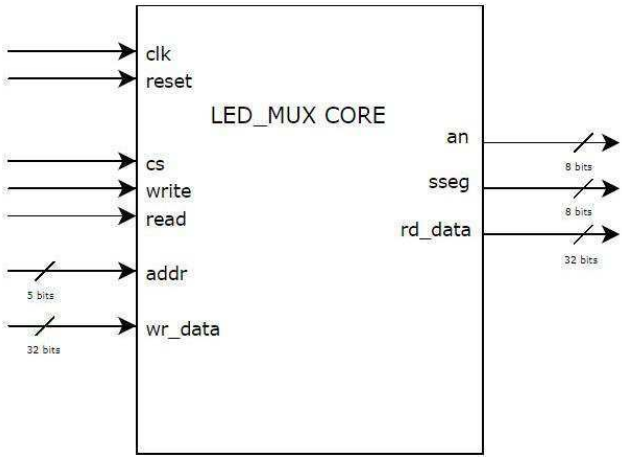


Ilustración 36: *Composición de E/S bloque LED-Mux*

5.7.2.1 Circuito de multiplexación de pantalla LED de 7 segmentos y ocho dígitos

El código para el circuito de multiplexación de ocho dígitos extendido se muestra en el Listado 7.7.

El diseño utiliza los tres MSB (bits de mayor peso) del contador para realizar una multiplexación de ocho a uno y generar la señal de activación a nivel bajo (0).

Listado 7.7. Circuito multiplexor para un display de 7 segmentos de 8 digitos

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity led_mux8 is
   port(
      clk, reset         : in  std_logic;
      in3, in2, in1, in0 : in  std_logic_vector(7 downto 0);
      in7, in6, in5, in4 : in  std_logic_vector(7 downto 0);
      an                 : out std_logic_vector(7 downto 0);
      sseg               : out std_logic_vector(7 downto 0)
   );
end led_mux8;

architecture arch of led_mux8 is
   -- refreshing rate around 1600 Hz (100MHz/2^16)
   constant N           : integer := 18;
   signal q_reg, q_next : unsigned(N - 1 downto 0);
   signal sel           : std_logic_vector(2 downto 0);
begin
   -- register
   process(clk, reset)
   begin
      if reset = '1' then
         q_reg <= (others => '0');
      elsif (clk'event and clk = '1') then
         q_reg <= q_next;
      end if;
   end process;
   -- next-state logic for the counter
   q_next <= q_reg + 1;
   -- 3 MSBs of counter to control 8-to-1 multiplexing
   -- and to generate active-low enable signal
   sel <= std_logic_vector(q_reg(N - 1 downto N - 3));
```

```
process(sel, in0, in1, in2, in3, in4, in5, in6, in7)
begin
      case sel is
         when "000" =>
            an   <= "11111110";
            sseg <= in0;
         when "001" =>
            an   <= "11111101";
            sseg <= in1;
         when "010" =>
            an   <= "11111011";
            sseg <= in2;
         when "011" =>
            an   <= "11110111";
            sseg <= in3;
         when "100" =>
            an   <= "11101111";
            sseg <= in4;
         when "101" =>
            an   <= "11011111";
            sseg <= in5;
         when "110" =>
            an   <= "10111111";
            sseg <= in6;
         when others =>
            an   <= "01111111";
            sseg <= in7;
      end case;
   end process;
end arch;
```

5.7.2.2 Mapa de registro y circuito de envoltura de ranura.

El circuito de multiplexación de ocho dígitos contiene ocho puertos de entrada de 8 bits, cada uno representa un patrón LED de 7 segmentos más el punto decimal. Los agrupamos en dos palabras de 32 bits. El mapa de registro de LED-MUX core se puede definir en consecuencia. Hay dos registros de escritura. Sus desplazamientos y campos de dirección son:

- Offset 0 (registro de datos de salida para los cuatro dígitos inferiores del display de 7 segmentos)
    - Bits 7 a 0: activación de LEDs del dígito 0 (dígito más a la derecha en la tarjeta Nexys 4 DDR).
    - Bits 15 a 8: activación de LEDs del dígito 1.
    - Bits 23 a 16: activación de LEDs del 2.
    - Bits 31 a 24: activación de LEDs del dígito 3.

- Offset 1 (registro de datos de salida para los cuatro dígitos superiores de la pantalla LED de siete segmentos)
    - Bits 7 a 0: activación de LEDs del dígito 4.
    - Bits 15 a 8: activación de LEDs del dígito 5.
    - Bits 23 a 16: activación de LEDs del dígito 6.
    - Bits 31 a 24: activación de LEDs del dígito 7.

Sobre la base del mapa de registro, podemos derivar un circuito de envoltura que cumpla con la especificación de la ranura y crear el bloque antirrebote. El código HDL del bloque se muestra en el Listado 7.8. Consta de dos registros de datos de salida para almacenar los ocho patrones y un circuito de decodificación de escritura.

Listado 7.8. Bloque LED-MUX

```
library ieee;
use ieee.std_logic_1164.all;

entity chu_led_mux_core is
   port(
      clk     : in  std_logic;
      reset   : in  std_logic;
      -- io bridge interface
      cs      : in  std_logic;
      write   : in  std_logic;
      read    : in  std_logic;  -- not used
      addr    : in  std_logic_vector(4 downto 0);
      rd_data : out std_logic_vector(31 downto 0);
      wr_data : in  std_logic_vector(31 downto 0);
      -- external interface
      an      : out std_logic_vector(7 downto 0);
      sseg    : out std_logic_vector(7 downto 0)
   );
end chu_led_mux_core;

architecture arch of chu_led_mux_core is
   signal d0_reg, d1_reg : std_logic_vector(31 downto 0);
   signal wr_en          : std_logic;
   signal wr_d0          : std_logic;
   signal wr_d1          : std_logic;
begin
   -- instantiate LED multiplexing circuit
   led_mux_unit : entity work.led_mux8
      port map(
         clk   => clk,
         reset => reset,
         in7   => d1_reg(31 downto 24),
         in6   => d1_reg(23 downto 16),
         in5   => d1_reg(15 downto 8),
         in4   => d1_reg(7 downto 0),
         in3   => d0_reg(31 downto 24),
         in2   => d0_reg(23 downto 16),
         in1   => d0_reg(15 downto 8),
         in0   => d0_reg(7 downto 0),
         an    => an,
         sseg  => sseg
      );

   -- 2 write registers
   process(clk, reset)
```

```
   begin
      if reset = '1' then
         d0_reg <= (others => '0');
         d1_reg <= (others => '1');
      elsif (clk'event and clk = '1') then
         if wr_d0 = '1' then
            d0_reg <= wr_data(31 downto 0);
         end if;
         if wr_d1 = '1' then
            d1_reg <= wr_data(31 downto 0);
         end if;
      end if;
   end process;
   -- decoding circuit
   wr_en <= '1' when write = '1' and cs = '1' else '0';
   wr_d0 <= '1' when addr(0) = '0' and wr_en = '1' else '0';
   wr_d1 <= '1' when addr(0) = '1' and wr_en = '1' else '0';
   -- unused
   rd_data <= (others => '0');
end arch;
```

5.7.2.3 Controlador LED-MUX

El controlador del LED-MUX core contiene métodos para escribir los patrones deseados en el display de 7 segmentos de ocho dígitos. La definición de clase se muestra en el Listado 7.9.

Listado 7.9. Definición de clase `SsegCore` *(*`sseg_core.h`*)*

```
#ifndef _SSEG_CORE_H_INCLUDED
#define _SSEG_CORE_H_INCLUDED

#include "chu_init.h"

class SsegCore {
public:
   enum {
      DATA_LOW_REG = 0, /**< 32-bit data for right 4 digits */
      DATA_HIGH_REG = 1 /**<  32-bit data for left 4 digits */
   };

   SsegCore(uint32_t core_base_addr);
   ~SsegCore(); // not used

   uint8_t h2s(int hex);
   void write_1ptn(uint8_t pattern, int pos);
   void write_8ptn(uint8_t *ptn_array);
   void set_dp(uint8_t pt);
private:
   uint32_t base_addr;
   uint8_t ptn_buf[8];    // led pattern buffer
   uint8_t dp;            // decimal point
   /* methods */
   void write_led();      // write patterns to reg
};
#endif  // _SSEG_CORE_H_INCLUDED
```

En la Ilustración 37 se repite un display de 7 segmentos de un solo dígito. Para facilitar el procesamiento, almacenamos los patrones de los LEDs en dos variables privadas. La variable `ptn_buf [8]` es una matriz de ocho elementos que almacena ocho patrones de LED de 7 segmentos. Un elemento tiene ocho bits de ancho y se define como *0gfedcba*, donde *g, f, e, d, c, b,* y *a* son los segmentos. La pantalla de la tarjeta Nexys 4 DDR está configurada activa a nivel bajo y, por lo tanto, se enciende un segmento LED cuando el valor

correspondiente es 0. La variable `dp` almacena los ocho puntos decimales. Se activa un punto decimal si el bit correspondiente en `dp` es 0. El método privado `write_led()` combina y empaqueta los datos de las dos variables en dos palabras de 32 bits y escribe las dos palabras en el núcleo LED-MUX.

El método `write_1ptn()` establece un patrón para un dígito específico de la pantalla, el método `write_8ptn()` establece los ocho dígitos, y el método `set_dp()` especifica los puntos decimales. El método `h2s()` convierte un dígito hexadecimal en un patrón de siete segmentos definido en la Ilustración 38.

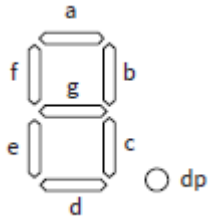


Ilustración 37: *Pantalla LED de siete segmentos.*

Ilustración 38: *Patrones hexadecimales.*

La implementación de la clase se muestra en el Listado 7.10.

Listado 7.10. Implementación de clase `SsegCore` (`sseg_core.cpp`)

```
#include "sseg_core.h"
SsegCore::SsegCore(uint32_t core_base_addr) {
   // pattern for "HI"; the order in array is reversed in 7-seg display
   // i.e., HI_PTN[0] is the leftmost led
   const uint8_t HI_PTN[]={0xff,0xf9,0x89,0xff,0xff,0xff,0xff,0xff};
   base_addr = core_base_addr;
   write_8ptn((uint8_t*) HI_PTN);
   set_dp(0x02);
}
SsegCore::~SsegCore() {
}
// not used
void SsegCore::write_led() {
   int i, p;
   uint32_t word = 0;

   // pack left 4 patterns into a 32-bit word
   // ptn_buf[0] is the leftmost led
   for (i = 0; i < 4; i++) {
      word = (word << 8) | ptn_buf[3 - i];
   }
   // incorporate decimal points (bit 7 of pattern)
   for (i = 0; i < 4; i++) {
      p = bit_read(dp, i);
      bit_write(word, 7 + 8 * i, p);
   }
```

```
   io_write(base_addr, DATA_LOW_REG, word);
   // pack right 4 patterns into a 32-bit word
   for (i = 0; i < 4; i++) {
      word = (word << 8) | ptn_buf[7 - i];
   }
   // incorporate decimal points
   for (i = 0; i < 4; i++) {
      p = bit_read(dp, 4 + i);
      bit_write(word, 7 + 8 * i, p);
   }
   io_write(base_addr, DATA_HIGH_REG, word);
}

void SsegCore::write_8ptn(uint8_t *ptn_array) {
   int i;

   for (i = 0; i < 8; i++) {
      ptn_buf[i] = *ptn_array;
      ptn_array++;
   }
   write_led();
}

void SsegCore::write_1ptn(uint8_t pattern, int pos) {
   ptn_buf[pos] = pattern;
   write_led();
}

// set decimal points,
// bits turn on the corresponding decimal points
void SsegCore::set_dp(uint8_t pt) {
   dp = ~pt;     // active low
   write_led();
}

// convert a hex digit to
uint8_t SsegCore::h2s(int hex) {
   /* active-low hex digit 7-seg patterns (0-9,a-f); MSB assigned to 1 */
   static const uint8_t PTN_TABLE[16] =
     {0xc0, 0xf9, 0xa4, 0xb0, 0x99, 0x92, 0x82, 0xf8, 0x80, 0x90, //0-9
      0x88, 0x83, 0xc6, 0xa1, 0x86, 0x8e };                       //a-f
   uint8_t ptn;

   if (hex < 16)
      ptn = PTN_TABLE[hex];
   else
      ptn = 0xff;
   return (ptn);
}
```

El constructor almacena la dirección base y muestra un patrón "*TFG 2019*" en la pantalla LED. Los patrones hexadecimales que no se encuentran en la Ilustración 38, han sido añadiodos del mismo modo que los existentes. Los métodos `write_1ptn()`, `write_8ptn()` y `set_dp()` primero actualizan las variables privadas correspondientes y luego llaman a `write_led()` para descargar los patrones al bloque para su visualización. El método `h2s()` usa una tabla de búsqueda de 16 elementos para almacenar los patrones de 7 segmentos para hacer la conversión. El método `write_led()` agrupa cuatro patrones de 8 bits en una palabra y luego incorpora los puntos decimale. Después escribe la palabra de 32 bits en un registro. La operación se repite dos veces para ocho dígitos.

6. Resultados finales

6.1 Introducción

Como parte de los objetivos de este Trabajo Fin de Grado, se contempla una validación de forma práctica de la metodología y funcionalidad del sistema SoC descrito, de manera que resulte sencillo, tomando esta demo como punto de partida, realizar proyectos y desarrollar aplicaciones más complejas, partiendo de unos conocimientos básicos de microcontroladores y programación en C y VHDL.

Como proyecto tipo demo, se ha realizado un programa para mostrar el funcionamiento de distintos bloques de los estudiados en la parte de *Análisis de Soluciones*.

La demo hace uso de varias de las opciones de componentes que facilita la propia tarjeta Nexys 4 DDR así como de algunos de sus periféricos. Los componentes y periféricos usados son los siguientes:

- Switches o interruptores.

- Conjunto de 16 LEDs.

- Comunicación serie por medio del bloque UART (Transmisor-Receptor síncrono Universal).

- Bloque PWM (modulación por ancho de pulso) para el funcionamiento de LEDs de color RGB.

- Puerto de señal analógica Pmod para el uso del periférico XADC (convertidor de analógico a digital) para mostrar la temperatura de la propia tarjeta Nexys 4 DDR.

- Conjunto de 5 pulsadores o botones haciendo uso del bloque Antirrebote.

- Display o pantalla de ocho dígitos de 7 segmentos.

Para conseguir el buen funcionamiento de los componentes y periféricos descritos, se ha realizado el diseño de una parte hardware (lenguaje HDL), programada por medio de la versión 2018.3 de Vivado (Xilinx Vivado Design Suite) y una parte software (lenguaje C/C++) con ayuda de la versión 2018.3 de SDK (Xilinx Vivado).

6.2 Parte Hardware

Esta parte implica la estructura principal del proyecto, y la unión entre todos los bloques que lo componen para así conseguir un funcionamiento exitoso del proyecto.

El bloque principal de la jerarquía, en el que se definen todas las entradas y salidas del proyecto y todos los componentes que se utilizan, es el llamado *“TFG_demo_final”*, archivo basado en el *mcs_top_sampler.vhd* visto en el apartado de Análisis de Soluciones.

El código de la definición de las distintas entradas y salidas que componen el proyecto principal, es decir, las que son utilizadas por el usuario es el siguiente:

```
 entity TFG_demo_final is
   generic(BRIDGE_BASE : std_logic_vector(31 downto 0) := x"C0000000");
   port(
      clk         : in    std_logic;
      reset_n     : in    std_logic;
      -- switches and LEDs
      sw          : in    std_logic_vector(15 downto 0);
      led         : out   std_logic_vector(15 downto 0);
      -- uart
      rx          : in    std_logic;
      tx          : out   std_logic;
      -- xadc
      adc_p       : in    std_logic_vector(3 downto 0);
      adc_n       : in    std_logic_vector(3 downto 0);
      -- rgb leds
      rgb_led1    : out   std_logic_vector(2 downto 0);
      rgb_led2    : out   std_logic_vector(2 downto 0);
      -- buttons
      btn         : in    std_logic_vector(4 downto 0);
      -- 8-digit 7-seg LEDs
      an          : out   std_logic_vector(7 downto 0);
      sseg        : out   std_logic_vector(7 downto 0)

   );
end TFG_demo_final;
Después de definir las entradas y salidas del proyecto, se definen las partes que lo
   componen como se muestra a continuación:
   component cpu
      port(
         clk             : in  std_logic;
         reset           : in  std_logic;
         io_addr_strobe  : out std_logic;
         io_read_strobe  : out std_logic;
         io_write_strobe : out std_logic;
         io_address      : out std_logic_vector(31 downto 0);
         io_byte_enable  : out std_logic_vector(3 downto 0);
         io_write_data   : out std_logic_vector(31 downto 0);
         io_read_data    : in  std_logic_vector(31 downto 0);
         io_ready        : in  std_logic
      );
   end component;
   component mmcm_fpro
      port(
         clk_in_100M : in  std_logic;
         clk_100M    : out std_logic;
         clk_25M     : out std_logic;
```

```
        clk_40M     : out std_logic;
        clk_67M     : out std_logic;
        reset       : in  std_logic;
        locked      : out std_logic
    );
  end component;

  signal io_addr_strobe  : std_logic;
  signal io_read_strobe  : std_logic;
  signal io_write_strobe : std_logic;
  signal io_byte_enable  : std_logic_vector(3 downto 0);
  signal io_address      : std_logic_vector(31 downto 0);
  signal io_write_data   : std_logic_vector(31 downto 0);
  signal io_read_data    : std_logic_vector(31 downto 0);
  signal io_ready        : std_logic;
  signal mmio_cs         : std_logic;
  signal mmio_wr         : std_logic;
  signal mmio_rd         : std_logic;
  signal mmio_addr       : std_logic_vector(20 downto 0);
  signal mmio_wr_data    : std_logic_vector(31 downto 0);
  signal mmio_rd_data    : std_logic_vector(31 downto 0);
  -- clk/reset related
  signal clk_100M        : std_logic;
  signal clk_25M         : std_logic;
  signal reset_sys       : std_logic;
  signal locked          : std_logic;
  -- pwm
  signal pwm             : std_logic_vector(7 downto 0);
  -- ddfs/audio pdm
  signal pdm             : std_logic;
  signal ddfs_sq_wave    : std_logic;
  -- fpro bus
  signal fp_wr             : std_logic;
  signal fp_addr           : std_logic_vector(20 downto 0);
  signal fp_wr_data        : std_logic_vector(31 downto 0);
  signal fp_video_cs       : std_logic;

begin
  -- clock and reset
  reset_sys <= (not locked) or (not reset_n);
  --  rgb leds
  rgb_led2           <= pwm(5 downto 3);
  rgb_led1           <= pwm(2 downto 0);
  -- instantiate clock management unit
  clk_mmcm_unit : mmcm_fpro
    port map(
      -- Clock in ports
      clk_in_100M => clk,
      -- Clock out ports
      clk_100M    => clk_100M,
      clk_25M     => clk_25M,
      clk_40M     => open,
      clk_67M     => open,
      -- Status and control signals
      reset       => '0',
      locked      => locked
    );
  -- instantiate microBlaze MCS
  mcs_0 : cpu
    port map(
      clk             => clk_100M,
      reset           => reset_sys,
      io_addr_strobe  => io_addr_strobe,
      io_read_strobe  => io_read_strobe,
      io_write_strobe => io_write_strobe,
```

```
      io_byte_enable  => io_byte_enable,
      io_address      => io_address,
      io_write_data   => io_write_data,
      io_read_data    => io_read_data,
      io_ready        => io_ready
   );
-- instantiate MCS IO bus to FPro bus bridge
bridge_unit : entity work.chu_mcs_bridge
   generic map(BRG_BASE => BRIDGE_BASE)
   port map(
      io_addr_strobe  => io_addr_strobe,
      io_read_strobe  => io_read_strobe,
      io_write_strobe => io_write_strobe,
      io_byte_enable  => io_byte_enable,
      io_address      => io_address,
      io_write_data   => io_write_data,
      io_read_data    => io_read_data,
      io_ready        => io_ready,
      fp_video_cs     => open,
      fp_mmio_cs      => mmio_cs,
      fp_wr           => mmio_wr,
      fp_rd           => mmio_rd,
      fp_addr         => mmio_addr,
      fp_wr_data      => mmio_wr_data,
      fp_rd_data      => mmio_rd_data
   );
-- instantiate sampler MMIO subsystem
mmio_sys_unit : entity work.mmio_sys_sampler
   port map(
      clk          => clk_100M,
      reset        => reset_sys,
      mmio_cs      => mmio_cs,
      mmio_wr      => mmio_wr,
      mmio_rd      => mmio_rd,
      mmio_addr    => mmio_addr,
      mmio_wr_data => mmio_wr_data,
      mmio_rd_data => mmio_rd_data,
      sw           => sw,
      led          => led,
      rx           => rx,
      tx           => tx,
      adc_p        => adc_p,
      adc_n        => adc_n,
      pwm          => pwm,
      btn          => btn,
      an           => an,
      sseg         => sseg
   );
-- instantiate daisy video subsystem
video_sys_unit : entity work.video_sys_daisy
   generic map(
      CD              => 12,
      VRAM_DATA_WIDTH => 9)
   port map(
      clk_sys       => clk_100M,
      clk_25M       => clk_25M,
      reset_sys     => reset_sys,
      video_cs      => fp_video_cs,
      video_wr      => fp_wr,
      video_addr    => fp_addr,
      video_wr_data => fp_wr_data
   );
```

En el siguiente nivel de jerarquía, se encuentran cinco componentes, que a su vez cada uno tiene varios escalones inferiores. Esos cinco componentes corresponden a:

1.- Circuito de administración del reloj

2.- Bloque del procesador (propio del fabricante)

3.- Puente de MCS a FPro

4.- Bloque del subsistema MMIO

5.- Bloque del subsistema de Video

A continuación, se van a explicar de forma detallada cada uno de los componentes anteriores.

6.2.1 Circuito de gestión del reloj

Se trata de un bloque propio del proveedor de Xilinx que se encarga de generar una serie de señales de reloj a distintas frecuencias para ser usadas durante el proyecto por los distintos bloques o componentes según sean necesarias.

Este archivo principal, en el cual están declarados los puertos de entrada y salida de las señales de reloj y las señales de estado y control de los mismos, se apoya en un segundo archivo, de un escalon inferior en la jerarquía, que sirve de memoria interna y es donde se guardan las distintas señales de reloj, de estado y de control necesarias.

El código del archivo principal es el siguiente:

```
module mmcm_fpro
 (
 // Clock in ports
  input         clk_in_100M,
  // Clock out ports
  output        clk_100M,
  output        clk_25M,
  output        clk_40M,
  output        clk_67M,
  // Status and control signals
  input         reset,
  output        locked
 );
```

```
mmcm_fpro_clk_wiz inst
  (
 // Clock in ports
  .clk_in_100M(clk_in_100M),
  // Clock out ports
  .clk_100M(clk_100M),
  .clk_25M(clk_25M),
  .clk_40M(clk_40M),
  .clk_67M(clk_67M),
  // Status and control signals
  .reset(reset),
  .locked(locked)
  );
Endmodule
```

6.2.2 Bloque del procesador

Este bloque, como bien se ha explicado en el apartado 5.1.3.2., utiliza los núcleos del proveedor de Xilinx, al igual que el controlador de memoria, el búfer de línea y el circuito de administración de reloj descrito en el apartado anterior.

No se va a entrar en detalle en mostrar este bloque, ya que ha sido incluido directamente obtenido de la web de Xilinx y sin programar nada. Por lo que nos limitamos a enumerar los archivos que lo componen:

- Bloque principal, archivo cpu.xci.
- Programación de la cpu en lenguaje VHD, archivo *cpu.vhdl*.
- Conjunto de módulos interconectados que componen el resto del bloque, se trata de un archivo de diseño esquemático que se muestra a continuación para un mejor entendimiento:

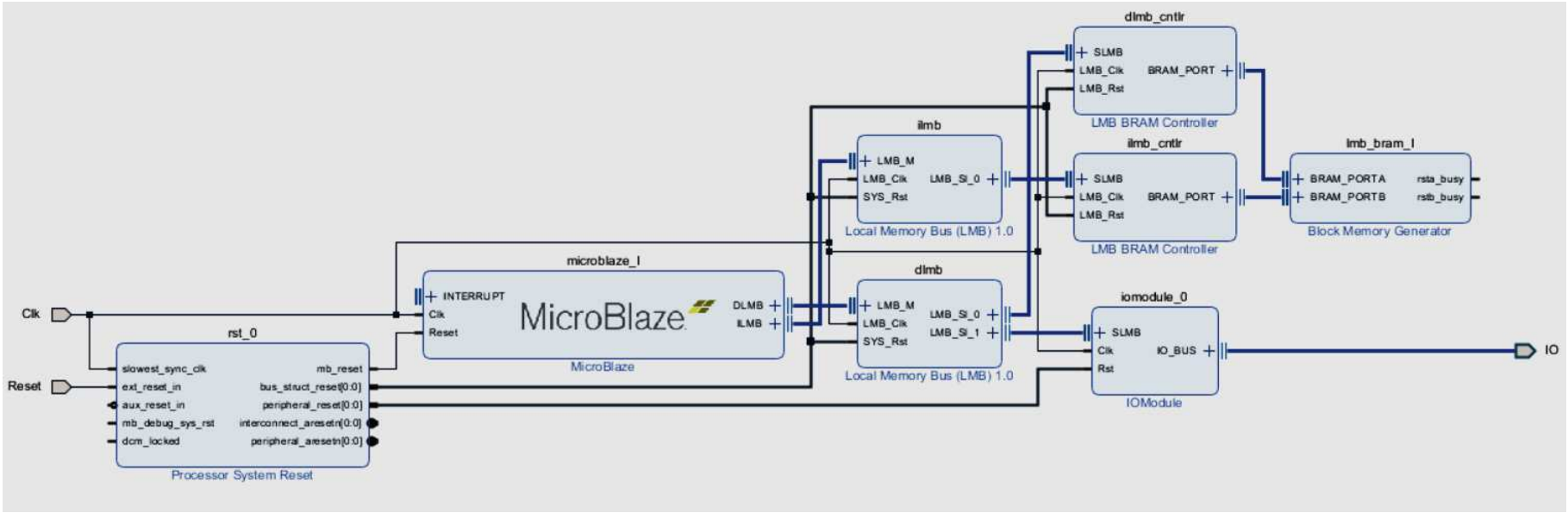


Ilustración 39: *Esquema del bloque del procesador (cpu.xci)*

Los archivos que componen el esquema de la Ilustración 39, viendo la imagen de frente y de izquierda a derecha son:

1.- Reset del sistema del procesador

2.- Chip de MicroBlaze

3.- Dos buses de memoria local

4.- Dos controladores de memoria RAM para los buses del punto anterior.

5.- Módulo de E/S

6.- Bloque generador de memoria

El conjunto formado por todos ellos, como muestra la ilustración anterior, componen el bloque completo más importante del proyecto, el procesador.

En los siguientes apartados, se describen los bloques de los componentes de entradas y salidas y periféricos utilizados en el proyecto, que junto al procesador y el circuito de administración de reloj, forman el segundo escalon de la jerarquía de este tipo de proyectos.

6.2.3 Puente MCS a FPro

Este módulo, como su propio nombre indica y como se ha estudiado anteriormente en la sección 5.3.6., realiza la adaptación o unión entre los dos buses distintos que componen el sistema: el bus de E/S MCS perteneciente al bloquedel procesador visto en el punto anterior y el bus FPro que corresponde a los subsistemas de MMIO y Video que siguen a continuación.

El circuito de traducción de direcciones decodifica la dirección de byte de 32 bits del bus de E/S MCS y la convierte en la dirección de palabra de 21 bits del bus FPro y las señales de selección de chip como muestra a continuación su código:

```
entity chu_mcs_bridge is
   generic(BRG_BASE : std_logic_vector(31 downto 0) := x"C0000000");
   port(
      -- uBlaze MCS I/O bus
      io_addr_strobe  : in  std_logic;  -- not used
      io_read_strobe  : in  std_logic;
      io_write_strobe : in  std_logic;
      io_byte_enable  : in  std_logic_vector(3 downto 0);
      io_address      : in  std_logic_vector(31 downto 0);
      io_write_data   : in  std_logic_vector(31 downto 0);
      io_read_data    : out std_logic_vector(31 downto 0);
      io_ready        : out std_logic;
```

```
      -- FPro bus
      fp_video_cs     : out std_logic;
      fp_mmio_cs      : out std_logic;
      fp_wr           : out std_logic;
      fp_rd           : out std_logic;
      fp_addr         : out std_logic_vector(20 downto 0);
      fp_wr_data      : out std_logic_vector(31 downto 0);
      fp_rd_data      : in  std_logic_vector(31 downto 0)
   );
end chu_mcs_bridge;

architecture arch of chu_mcs_bridge is
   signal mcs_bridge_en : std_logic;
   signal word_addr     : std_logic_vector(29 downto 0);
begin
   -- address translation and decoding
   -- 2 LSBs are "00" due to word alignment
   word_addr     <= io_address(31 downto 2);
   mcs_bridge_en <=
      '1' when io_address(31 downto 24)=BRG_BASE(31 downto 24) else '0';
   fp_video_cs   <=
      '1' when mcs_bridge_en='1' and io_address(23)='1' else '0';
   fp_mmio_cs    <=
      '1' when mcs_bridge_en='1' and io_address(23)='0' else '0';
   fp_addr       <= word_addr(20 downto 0);
   -- control line conversion
   fp_wr         <= io_write_strobe;
   fp_rd         <= io_read_strobe;
   io_ready      <= '1'; -- not used; transaction done in 1 clock
   -- data line conversion
   fp_wr_data    <= io_write_data;
   io_read_data  <= fp_rd_data;
end arch;
```

Una vez establecida la comunicación de los buses y adaptación del procesador con el resto de los componentes, se añaden todos los bloques de E/S y periféricos que se utilizan para el proyecto y que van conectados al bus FPro.

6.2.4 Bloque del subsistema MMIO

Este subsistema, contiene la gran mayoría de componentes y periféricos disponibles de la tarjeta Nexys 4 DDR.

6.2.4.1 Archivo principal

Contiene un archivo principal que define el subsistema MMIO, donde se declaran todas las E/S, así como cada uno de los accesos de memoria que componen el proyecto de la demo. El código es el que se muestra a continuación:

```
entity mmio_sys_sampler is
   port(
      -- FPro bus
      clk          : in    std_logic;
      reset        : in    std_logic;
      mmio_cs      : in    std_logic;
      mmio_wr      : in    std_logic;
      mmio_rd      : in    std_logic;
      mmio_addr    : in    std_logic_vector(20 downto 0);
      mmio_wr_data : in    std_logic_vector(31 downto 0);
      mmio_rd_data : out   std_logic_vector(31 downto 0);
      -- switches and LEDs
      sw           : in    std_logic_vector(15 downto 0);
```

```
      led          : out   std_logic_vector(15 downto 0);
      -- uart
      rx           : in    std_logic;
      tx           : out   std_logic;
      -- 4 analog input pair
      adc_p        : in    std_logic_vector(3 downto 0);
      adc_n        : in    std_logic_vector(3 downto 0);
      -- pwm
      pwm          : out   std_logic_vector(7 downto 0);
      -- btn
      btn          : in    std_logic_vector(4 downto 0);
      -- 8-digit 7-seg LEDs
      an           : out   std_logic_vector(7 downto 0);
      sseg         : out   std_logic_vector(7 downto 0)
   );
end mmio_sys_sampler;

architecture arch of mmio_sys_sampler is
   signal cs_array       : std_logic_vector(63 downto 0);
   signal reg_addr_array : slot_2d_reg_type;
   signal mem_rd_array   : std_logic_vector(63 downto 0);
   signal mem_wr_array   : std_logic_vector(63 downto 0);
   signal rd_data_array  : slot_2d_data_type;
   signal wr_data_array  : slot_2d_data_type;
   signal adsr_env       : std_logic_vector(15 downto 0);
begin
   --****************************************************************
   --  MMIO controller instantiation
   --****************************************************************
   ctrl_unit : entity work.chu_mmio_controller
      port map(
         -- FPro bus interface
         mmio_cs             => mmio_cs,
         mmio_wr             => mmio_wr,
         mmio_rd             => mmio_rd,
         mmio_addr           => mmio_addr,
         mmio_wr_data        => mmio_wr_data,
         mmio_rd_data        => mmio_rd_data,
         -- 64 slot interface
         slot_cs_array       => cs_array,
         slot_reg_addr_array => reg_addr_array,
         slot_mem_rd_array   => mem_rd_array,
         slot_mem_wr_array   => mem_wr_array,
         slot_rd_data_array  => rd_data_array,
         slot_wr_data_array  => wr_data_array
      );
   --****************************************************************
   -- IO slots instantiations
   --****************************************************************
   -- slot 0: system timer
   timer_slot0 : entity work.chu_timer
      port map(
         clk     => clk,
         reset   => reset,
         cs      => cs_array(S0_SYS_TIMER),
         read    => mem_rd_array(S0_SYS_TIMER),
         write   => mem_wr_array(S0_SYS_TIMER),
         addr    => reg_addr_array(S0_SYS_TIMER),
         rd_data => rd_data_array(S0_SYS_TIMER),
         wr_data => wr_data_array(S0_SYS_TIMER)
      );
   -- slot 1: uart1
   uart1_slot1 : entity work.chu_uart
      generic map(FIFO_DEPTH_BIT => 6)
      port map(
         clk     => clk,
         reset   => reset,
         cs      => cs_array(S1_UART1),
         read    => mem_rd_array(S1_UART1),
         write   => mem_wr_array(S1_UART1),
         addr    => reg_addr_array(S1_UART1),
         rd_data => rd_data_array(S1_UART1),
         wr_data => wr_data_array(S1_UART1),
         -- external signals
```

```
      tx      => tx,
      rx      => rx
   );
-- slot 2: GPO for 16 LEDs
gpo_slot2 : entity work.chu_gpo
   generic map(W => 16)
   port map(
      clk     => clk,
      reset   => reset,
      cs      => cs_array(S2_LED),
      read    => mem_rd_array(S2_LED),
      write   => mem_wr_array(S2_LED),
      addr    => reg_addr_array(S2_LED),
      rd_data => rd_data_array(S2_LED),
      wr_data => wr_data_array(S2_LED),
      -- external signal
      dout    => led
   );
-- slot 3: input port for 16 slide switches
gpi_slot3 : entity work.chu_gpi
   generic map(W => 16)
   port map(
      clk     => clk,
      reset   => reset,
      cs      => cs_array(S3_SW),
      read    => mem_rd_array(S3_SW),
      write   => mem_wr_array(S3_SW),
      addr    => reg_addr_array(S3_SW),
      rd_data => rd_data_array(S3_SW),
      wr_data => wr_data_array(S3_SW),
      -- external signal
      din     => sw
   );
-- slot 4: xadc
xadc_slot4 : entity work.chu_xadc_core
   port map(
      clk     => clk,
      reset   => reset,
      cs      => cs_array(S5_XADC),
      read    => mem_rd_array(S5_XADC),
      write   => mem_wr_array(S5_XADC),
      addr    => reg_addr_array(S5_XADC),
      rd_data => rd_data_array(S5_XADC),
      wr_data => wr_data_array(S5_XADC),
      -- external signal
      adc_p   => adc_p,
      adc_n   => adc_n
   );
-- slot 5: pwm
pwm_slot5 : entity work.chu_io_pwm_core
   generic map(
      W => 8,
      R => 10)
   port map(
      clk     => clk,
      reset   => reset,
      cs      => cs_array(S6_PWM),
      read    => mem_rd_array(S6_PWM),
      write   => mem_wr_array(S6_PWM),
      addr    => reg_addr_array(S6_PWM),
      rd_data => rd_data_array(S6_PWM),
      wr_data => wr_data_array(S6_PWM),
      -- external interface
      pwm_out => pwm
   );
-- slot 6: push button
debounce_slot6 : entity work.chu_debounce_core
   generic map(
      W => 5,
      N => 20
   )
   port map(
      clk     => clk,
      reset   => reset,
```

```
         cs      => cs_array(S7_BTN),
         read    => mem_rd_array(S7_BTN),
         write   => mem_wr_array(S7_BTN),
         addr    => reg_addr_array(S7_BTN),
         rd_data => rd_data_array(S7_BTN),
         wr_data => wr_data_array(S7_BTN),
         -- external interface
         din     => btn
      );
   -- slot 7: 7-seg LED
   sseg_led_slot7 : entity work.chu_led_mux_core
      port map(
         clk     => clk,
         reset   => reset,
         cs      => cs_array(S8_SSEG),
         read    => mem_rd_array(S8_SSEG),
         write   => mem_wr_array(S8_SSEG),
         addr    => reg_addr_array(S8_SSEG),
         rd_data => rd_data_array(S8_SSEG),
         wr_data => wr_data_array(S8_SSEG),
         -- external interface
         an      => an,
         sseg    => sseg
      );
```

Una vez definidos todos los espacios de memoria necesarios para la realización del proyecto, se procede a programar el funcionamiento de cada uno de ellos con los archivos necesarios.

6.2.4.2 Controlador

En este bloque, además de la programación de cada uno de los componentes usados para esta demo y los archivos derivados que cada uno contiene, se incluye como parte principal el controlador MMIO estudiado en el apartado 5.3.5.

Este controlador, realiza decodificación y multiplexación a nivel de subsistema y sirve como un circuito de interconexión entre el bus FPro y los núcleos de E/S utilizados. El código de programación del controlador MMIO utilizado para esta demo es el siguiente:

```
entity chu_mmio_controller is
   port(
      -- FPro bus
      mmio_cs             : in  std_logic;
      mmio_wr             : in  std_logic;
      mmio_rd             : in  std_logic;
      mmio_addr           : in  std_logic_vector(20 downto 0);
      mmio_wr_data        : in  std_logic_vector(31 downto 0);
      mmio_rd_data        : out std_logic_vector(31 downto 0);
      -- slot interface
      slot_cs_array       : out std_logic_vector(63 downto 0);
      slot_mem_rd_array   : out std_logic_vector(63 downto 0);
      slot_mem_wr_array   : out std_logic_vector(63 downto 0);
      slot_reg_addr_array : out slot_2d_reg_type;
      slot_rd_data_array  : in  slot_2d_data_type;
      slot_wr_data_array  : out slot_2d_data_type
   );
end chu_mmio_controller;
```

```
architecture arch of chu_mmio_controller is
   -- 11 LSBs of address used; 2^6 slots, each with 2^5 registers
   alias slot_addr : std_logic_vector(5 downto 0) is
         mmio_addr(10 downto 5);
   alias reg_addr  : std_logic_vector(4 downto 0) is
         mmio_addr(4 downto 0);
begin
   -- address decoding
      process(slot_addr, mmio_cs)
   begin
      slot_cs_array <= (others => '0');
      if mmio_cs = '1' then
         slot_cs_array(to_integer(unsigned(slot_addr))) <= '1';
      end if;
   end process;
   -- broadcast to all slots
   slot_mem_rd_array   <= (others => mmio_rd);
   slot_mem_wr_array   <= (others => mmio_wr);
   slot_wr_data_array  <= (others => mmio_wr_data);
   slot_reg_addr_array <= (others => reg_addr);
   -- mux for read data
   mmio_rd_data <= slot_rd_data_array(to_integer(unsigned(slot_addr)));
end arch;
```

A continuación, se van describiendo los códigos de programa de los archivos principales de cada uno de los componentes utilizados en el proyecto, indicando en la sección de este documento donde se encuentra su estudio y desarrollo:

6.2.4.3 Espacio de Memoria 0.- Núcleo Temporizador apartado 5.3.4.

```
entity chu_timer is
   port(
      clk     : in  std_logic;
      reset   : in  std_logic;
      -- slot interface
      cs      : in  std_logic;
      write   : in  std_logic;
      read    : in  std_logic;
      addr    : in  std_logic_vector(4 downto 0);
      rd_data : out std_logic_vector(31 downto 0);
      wr_data : in  std_logic_vector(31 downto 0)
   );
end chu_timer;

architecture arch of chu_timer is
   signal count_reg  : unsigned(47 downto 0);
   signal count_next : unsigned(47 downto 0);
   signal ctrl_reg   : std_logic;
   signal wr_en      : std_logic;
   signal clear, go  : std_logic;
begin
   --*****************************************************************
   -- counter
   --*****************************************************************
   -- register
   process(clk, reset)
   begin
      if reset = '1' then
         count_reg <= (others => '0');
      elsif (clk'event and clk = '1') then
         count_reg <= count_next;
      end if;
   end process;
   -- next-state logic
   count_next <= (others => '0') when clear = '1' else
                 count_reg + 1   when go = '1' else
                 count_reg;

   --*****************************************************************
```

```
   -- wrapping circuit
   --*****************************************************************
  -- ctrl register
   process(clk, reset)
   begin
      if reset = '1' then
         ctrl_reg <= '0';
      elsif (clk'event and clk = '1') then
         if wr_en = '1' then
            ctrl_reg <= wr_data(0);
         end if;
      end if;
   end process;
   -- decoding logic
   wr_en <=
      '1' when write='1' and cs='1' and addr(1 downto 0)="10" else '0';
   clear <= '1' when wr_en='1' and wr_data(1)='1' else '0';
   go    <= ctrl_reg;
   -- slot read multiplexing
   rd_data <=
      std_logic_vector(count_reg(31 downto 0)) when addr(0)='0' else
      x"0000" & std_logic_vector(count_reg(47 downto 32));
end arch;
```

6.2.4.4 Espacio de Memoria 1.- Núcleo UART apartado 5.4.

```
entity chu_uart is
   generic(
      FIFO_DEPTH_BIT : integer := 8 -- # FIFO addr bits
   );
   port(
      clk     : in  std_logic;
      reset   : in  std_logic;
      -- slot interface
      cs      : in  std_logic;
      write   : in  std_logic;
      read    : in  std_logic;
      addr    : in  std_logic_vector(4 downto 0);
      rd_data : out std_logic_vector(31 downto 0);
      wr_data : in  std_logic_vector(31 downto 0);
      -- external signals
      tx      : out std_logic;
      rx      : in  std_logic
   );
end chu_uart;

architecture arch of chu_uart is
   signal wr_en    : std_logic;
   signal wr_uart  : std_logic;
   signal rd_uart  : std_logic;
   signal wr_dvsr  : std_logic;
   signal tx_full  : std_logic;
   signal rx_empty : std_logic;
   signal r_data   : std_logic_vector(7 downto 0);
   signal dvsr_reg : std_logic_vector(10 downto 0);
begin
   -- instantiate uart controller
   uart_unit : entity work.uart(str_arch)
      generic map(
         DBIT    => 8,
         SB_TICK => 16,
         FIFO_W  => FIFO_DEPTH_BIT
      )
      port map(
         clk      => clk,
         reset    => reset,
         rd_uart  => rd_uart,
         wr_uart  => wr_uart,
         dvsr     => dvsr_reg,
         rx       => rx,
         tx       => tx,
         w_data   => wr_data(7 downto 0),
         r_data   => r_data,
```

```
         tx_full  => tx_full,
         rx_empty => rx_empty
      );
   -- baud rate register
   process(clk, reset)
   begin
      if (reset = '1') then
         dvsr_reg <= (others => '0');
      elsif (clk'event and clk = '1') then
         if wr_dvsr = '1' then
            dvsr_reg <= wr_data(10 downto 0);
         end if;
      end if;
   end process;
   -- write decoding
   wr_en   <= '1' when write = '1' and cs = '1' else '0';
   wr_dvsr <= '1' when addr(1 downto 0)="01" and wr_en = '1' else '0';
   wr_uart <= '1' when addr(1 downto 0)="10" and wr_en = '1' else '0';
   rd_uart <= '1' when addr(1 downto 0)="11" and wr_en = '1' else '0';
   -- read multiplexing
   rd_data <= x"00000" & "00" & tx_full & rx_empty & r_data;
end arch;
```

6.2.4.5 Espacio de Memoria 2.- Núcleo GPO apartado 5.3.3.

```
entity chu_gpo is
   generic(W : integer := 8);  -- width of output port
   port(
      clk     : in  std_logic;
      reset   : in  std_logic;
      -- slot interface
      cs      : in  std_logic;
      write   : in  std_logic;
      read    : in  std_logic;
      addr    : in  std_logic_vector(4 downto 0);
      rd_data : out std_logic_vector(31 downto 0);
      wr_data : in  std_logic_vector(31 downto 0);
      -- external signal
      dout    : out std_logic_vector(W - 1 downto 0)
   );
end chu_gpo;

architecture arch of chu_gpo is
   signal buf_reg : std_logic_vector(W - 1 downto 0);
   signal wr_en   : std_logic;
begin
   -- output buffer register
   process(clk, reset)
   begin
      if (reset = '1') then
         buf_reg <= (others => '0');
      elsif (clk'event and clk = '1') then
         if wr_en = '1' then
            buf_reg <= wr_data(W - 1 downto 0);
         end if;
      end if;
   end process;
   -- decoding logic
   wr_en   <= '1' when write = '1' and cs = '1' else '0';
   -- slot read interface
   rd_data <= (others => '0');          -- not used
   -- external output
   dout    <= buf_reg;
end arch;
```

6.2.4.6 Espacio de Memoria 3.- Núcleo GPI apartado 5.3.3.

```
entity chu_gpi is
   generic(W : integer := 8);  -- width of input port
   port(
      clk     : in  std_logic;
      reset   : in  std_logic;
      -- slot interface
      cs      : in  std_logic;
      write   : in  std_logic;
      read    : in  std_logic;
      addr    : in  std_logic_vector(4 downto 0);
      rd_data : out std_logic_vector(31 downto 0);
      wr_data : in  std_logic_vector(31 downto 0);
      -- external signal
      din     : in  std_logic_vector(W-1 downto 0)
   );
end chu_gpi;

architecture arch of chu_gpi is
   signal rd_data_reg : std_logic_vector(W-1 downto 0);
begin
   -- input register
   process(clk, reset)
   begin
      if reset = '1' then
         rd_data_reg <= (others => '0');
      elsif (clk'event and clk = '1') then
         rd_data_reg <= din;
      end if;
   end process;
   -- slot read interface
   rd_data(W-1 downto 0) <= rd_data_reg;
   rd_data(31 downto W)  <= (others => '0');
end arch;
Espacio de Memoria 4.- Núcleo XADC apartado 5.5.
entity chu_xadc_core is
   port(
      clk     : in  std_logic;
      reset   : in  std_logic;
      -- slot interface
      cs      : in  std_logic;
      write   : in  std_logic;
      read    : in  std_logic;
      addr    : in  std_logic_vector(4 downto 0);
      rd_data : out std_logic_vector(31 downto 0);
      wr_data : in  std_logic_vector(31 downto 0);
      -- external signals
      adc_p   : in  std_logic_vector(3 downto 0);
      adc_n   : in  std_logic_vector(3 downto 0)
   );
end chu_xadc_core;

architecture arch of chu_xadc_core is
   component xadc_fpro
      port(
         di_in       : in  std_logic_vector(15 downto 0);
         daddr_in    : in  std_logic_vector(6 downto 0);
         den_in      : in  std_logic;
         dwe_in      : in  std_logic;
         drdy_out    : out std_logic;
         do_out      : out std_logic_vector(15 downto 0);
         dclk_in     : in  std_logic;
         reset_in    : in  std_logic;
         vp_in       : in  std_logic;
         vn_in       : in  std_logic;
         vauxp2      : in  std_logic;
         vauxn2      : in  std_logic;
         vauxp3      : in  std_logic;
```

```
            vauxn3      : in  std_logic;
            vauxp10     : in  std_logic;
            vauxn10     : in  std_logic;
            vauxp11     : in  std_logic;
            vauxn11     : in  std_logic;
            channel_out : out std_logic_vector(4 downto 0);
            eoc_out     : out std_logic;
            alarm_out   : out std_logic;
            eos_out     : out std_logic;
            busy_out    : out std_logic
         );
      end component;
      signal channel      : std_logic_vector(4 downto 0);
      signal daddr_in     : std_logic_vector(6 downto 0);
      signal eoc          : std_logic;
      signal rdy          : std_logic;
      signal adc_data     : std_logic_vector(15 downto 0);
      signal adc0_out_reg : std_logic_vector(15 downto 0);
      signal adc1_out_reg : std_logic_vector(15 downto 0);
      signal adc2_out_reg : std_logic_vector(15 downto 0);
      signal adc3_out_reg : std_logic_vector(15 downto 0);
      signal tmp_out_reg  : std_logic_vector(15 downto 0);
      signal vcc_out_reg  : std_logic_vector(15 downto 0);
   begin
      -- instantiate cusomized xadc core
      xdac_unit : xadc_fpro
         port map(
            dclk_in     => clk,
            reset_in    => reset,          --reset,
            di_in       => (others => '0'),
            daddr_in    => daddr_in,
            den_in      => eoc,
            dwe_in      => '0',            -- read only
            drdy_out    => rdy,
            do_out      => adc_data,
            vp_in       => '0',
            vn_in       => '0',
            vauxp2      => adc_p(2),
            vauxn2      => adc_n(2),
            vauxp3      => adc_p(0),
            vauxn3      => adc_n(0),
            vauxp10     => adc_p(1),
            vauxn10     => adc_n(1),
            vauxp11     => adc_p(3),
            vauxn11     => adc_n(3),
            channel_out => channel,
            eoc_out     => eoc,
            eos_out     => open,
            busy_out    => open,
            alarm_out   => open
         );
      -- form xadc DRP address
      daddr_in <= "00" & channel;
      -- registers and decoding
      process(clk, reset)
      begin
         if reset = '1' then
            adc0_out_reg <= (others => '0');
            adc1_out_reg <= (others => '0');
            adc2_out_reg <= (others => '0');
            adc3_out_reg <= (others => '0');
            tmp_out_reg  <= (others => '0');
            vcc_out_reg  <= (others => '0');
```

```
       elsif (clk'event and clk = '1') then
         if rdy = '1' and channel = "10011" then
            adc0_out_reg <= adc_data;
         end if;
         if rdy = '1' and channel = "11010" then
            adc1_out_reg <= adc_data;
         end if;
         if rdy = '1' and channel = "10010" then
            adc2_out_reg <= adc_data;
         end if;
         if rdy = '1' and channel = "11011" then
            adc3_out_reg <= adc_data;
         end if;
         if rdy = '1' and channel = "00000" then
            tmp_out_reg <= adc_data;
         end if;
         if rdy = '1' and channel = "00001" then
            vcc_out_reg <= adc_data;
         end if;
      end if;
   end process;
   -- read multiplexing
   with addr(2 downto 0) select
      rd_data <=
         x"0000" & adc0_out_reg when "000",
         x"0000" & adc1_out_reg when "001",
         x"0000" & adc2_out_reg when "010",
         x"0000" & adc3_out_reg when "011",
         x"0000" & tmp_out_reg  when "100",
         x"0000" & vcc_out_reg  when others;
end arch;
Espacio de Memoria 5.- Núcleo PWM apartado 5.6.
entity chu_io_pwm_core is
   generic(
      W : integer := 8;    -- width (# bits) of output port
      R : integer := 8     -- # bits of PWM resolution (2^R levels)
   );
   port(
      clk     : in  std_logic;
      reset   : in  std_logic;
      -- slot interface
      cs      : in  std_logic;
      write   : in  std_logic;
      read    : in  std_logic;
      addr    : in  std_logic_vector(4 downto 0);
      rd_data : out std_logic_vector(31 downto 0);
      wr_data : in  std_logic_vector(31 downto 0);
      -- external signals
      pwm_out : out std_logic_vector(W - 1 downto 0)
   );
end chu_io_pwm_core;

architecture arch of chu_io_pwm_core is
   type reg_file_type is array (W - 1 downto 0) of
        std_logic_vector(R downto 0);
   signal duty_2d_reg    : reg_file_type;
   signal wr_en, dvsr_en : std_logic;
   signal duty_array_en  : std_logic;
   signal q_reg, q_next  : unsigned(31 downto 0);
   signal d_reg, d_next  : unsigned(R - 1 downto 0);
   signal d_ext          : unsigned(R downto 0);
   signal pwm_next       : std_logic_vector(W - 1 downto 0);
   signal pwm_reg        : std_logic_vector(W - 1 downto 0);
   signal tick           : std_logic;
   signal dvsr_reg       : std_logic_vector(31 downto 0);
```

```
begin
   --*****************************************************************
   -- wrapping circuit
   --*****************************************************************
   --  decoding logic
   wr_en         <= '1' when write = '1' and cs = '1' else '0';
   duty_array_en <= '1' when wr_en = '1' and addr(4) = '1' else '0';
   dvsr_en       <= '1' when wr_en = '1' and addr = "00000" else '0';
   -- register for divisor
   process(clk, reset)
   begin
      if (reset = '1') then
         dvsr_reg <= (others => '0');
      elsif (clk'event and clk = '1') then
         if dvsr_en = '1' then
            dvsr_reg <= wr_data;
         end if;
      end if;
   end process;
   -- register file for duty cycles
   process(clk, reset)
   begin
      if (reset = '1') then
         duty_2d_reg <= (others => (others => '0'));
      elsif (clk'event and clk = '1') then
         if duty_array_en = '1' then
           duty_2d_reg(to_integer(unsigned(addr(3 downto 0))))<=wr_data;
         end if;
      end if;
   end process;
   --*****************************************************************
   -- multi-bit PWM
   --*****************************************************************
   process(clk, reset)
   begin
      if reset = '1' then
         q_reg   <= (others => '0');
         d_reg   <= (others => '0');
         pwm_reg <= (others => '0');
      elsif (clk'event and clk = '1') then
         q_reg   <= q_next;
         d_reg   <= d_next;
         pwm_reg <= pwm_next;
      end if;
   end process;
   -- "prescale" counter
   q_next <= (others=>'0') when q_reg=unsigned(dvsr_reg) else q_reg + 1;
   tick   <= '1' when q_reg = 0 else '0';
   -- duty cycle counter
   d_next <= d_reg + 1 when tick = '1' else d_reg;
   d_ext  <= '0' & d_reg;
   -- comparison circuit
   gen_comp_cell : for i in 0 to W - 1 generate
      pwm_next(i) <= '1' when d_ext<unsigned(duty_2d_reg(i)) else '0';
   end generate;
   pwm_out <= pwm_reg;
   -- read data not used
   rd_data <= (others => '0');
end arch;
Espacio de Memoria 6.- Núcleo Antirrebote apartado 5.7.1.
entity chu_debounce_core is
   generic(
      W : integer := 8;    -- width of input port
      N : integer := 20    -- # bit for 10-ms tick 2^N * clk period
   );
```

```
   port(clk     : in  std_logic;
        reset   : in  std_logic;
        -- io bridge interface
        cs      : in  std_logic;
        write   : in  std_logic;
        read    : in  std_logic;
        addr    : in  std_logic_vector(4 downto 0);
        rd_data : out std_logic_vector(31 downto 0);
        wr_data : in  std_logic_vector(31 downto 0);
        -- external signal
        din     : in  std_logic_vector(W-1 downto 0));
end chu_debounce_core;

architecture arch of chu_debounce_core is
   signal rd_data_reg : std_logic_vector(W-1 downto 0);
   signal m_tick      : std_logic;
   signal db_out      : std_logic_vector(W-1 downto 0);
begin
   --**************************************************************
   -- input register
   --**************************************************************
   process(clk, reset)
   begin
      if reset = '1' then
         rd_data_reg <= (others => '0');
      elsif (clk'event and clk = '1') then
         rd_data_reg <= din;
      end if;
   end process;
   --**************************************************************
   -- instantiate one counter and W debouncing FSMs
   --**************************************************************
   db_counter_unit : entity work.debounce_counter
      generic map(N => N)
      port map(
         clk    => clk,
         reset  => reset,
         m_tick => m_tick
      );
   gen_fsm_cell : for i in 0 to W-1 generate
      db_fsm_unit : entity work.debounce_fsm
         port map(
            clk    => clk,
            reset  => reset,
            sw     => din(i),
            m_tick => m_tick,
            db     => db_out(i)
         );
   end generate;
   --**************************************************************
   -- read multiplexing
   --**************************************************************
   rd_data(W-1 downto 0) <= rd_data_reg when addr(0)='0' else db_out;
   rd_data(31 downto W)  <= (others => '0');
end arch;
```

6.2.4.7 Espacio de Memoria 0.- Núcleo Pantalla 7 segmentos apartado 5.7.2.

```
entity chu_led_mux_core is
   port(
      clk     : in  std_logic;
      reset   : in  std_logic;
      -- io bridge interface
      cs      : in  std_logic;
      write   : in  std_logic;
      read    : in  std_logic;  -- not used
      addr    : in  std_logic_vector(4 downto 0);
      rd_data : out std_logic_vector(31 downto 0);
      wr_data : in  std_logic_vector(31 downto 0);
      -- external interface
      an      : out std_logic_vector(7 downto 0);
      sseg    : out std_logic_vector(7 downto 0)
   );
end chu_led_mux_core;

architecture arch of chu_led_mux_core is
   signal d0_reg, d1_reg : std_logic_vector(31 downto 0);
   signal wr_en          : std_logic;
   signal wr_d0          : std_logic;
   signal wr_d1          : std_logic;
begin
   -- instantiate LED multiplexing circuit
   led_mux_unit : entity work.led_mux8
      port map(
         clk   => clk,
         reset => reset,
         in7   => d1_reg(31 downto 24),
         in6   => d1_reg(23 downto 16),
         in5   => d1_reg(15 downto 8),
         in4   => d1_reg(7 downto 0),
         in3   => d0_reg(31 downto 24),
         in2   => d0_reg(23 downto 16),
         in1   => d0_reg(15 downto 8),
         in0   => d0_reg(7 downto 0),
         an    => an,
         sseg  => sseg
      );

   -- 2 write registers
   process(clk, reset)
   begin
      if reset = '1' then
         d0_reg <= (others => '0');
         d1_reg <= (others => '1');
      elsif (clk'event and clk = '1') then
         if wr_d0 = '1' then
            d0_reg <= wr_data(31 downto 0);
         end if;
         if wr_d1 = '1' then
            d1_reg <= wr_data(31 downto 0);
         end if;
      end if;
   end process;
   -- decoding circuit
   wr_en <= '1' when write = '1' and cs = '1' else '0';
   wr_d0 <= '1' when addr(0) = '0' and wr_en = '1' else '0';
   wr_d1 <= '1' when addr(0) = '1' and wr_en = '1' else '0';
   -- unused
   rd_data <= (others => '0');
end arch;
```

Cabe mencionar que el código mostrado en estos últimos puntos, corresponde solamente a los archivos principales de cada bloque o núcleo, algunos de estos bloques contienen archivos derivados de los principales que son integrados de forma completa en el formato digital de este trabajo.

6.2.5 Bloque del subsistema de Video

El bloque de componentes de video y uso del periférico de VGA no ha podido ser incluido en este proyecto por motivos de tiempo.

6.3 Parte software

En cuanto a la programación software por medio de lenguaje C/C++ para esta demo, consta de una serie de archivos que en su conjunto hacen que todos los periféricos utilizados para la demo con las entradas y salidas correspondientes funcionen correctamente, como se ha desarrollado en la sección 5.2.

A continuación, por orden de prioridad según el proyecto, se van a ir describiendo los distintos archivos software utilizados, su función y parte de su código.

6.3.1 Chu_io_map.h

Mantiene la información de mapeo de direcciones para el código C/C++. Realiza la misma función que el archivo hardware *chu_io_map.vhd*. El código es el siguiente:

```
#ifndef _CHU_IO_MAP_INCLUDED
#define _CHU_IO_MAP_INCLUDED

#ifdef __cplusplus
extern "C" {
#endif

/*****************************************************************
 * Xilinx nexys4 ddr board "sampler/daisy" configuration
 *****************************************************************/
// #ifdef _NEXYS4
// system clock rate in MHz; used for timer and uart
#define SYS_CLK_FREQ 100

//io base address for microBlaze MCS
#define BRIDGE_BASE 0xc0000000

// slot module definition
// format: Slot#_ModuleType_Name
#define S0_SYS_TIMER  0
#define S1_UART1      1
#define S2_LED        2
#define S3_SW         3
#define S4_USER       4
#define S5_XDAC       5
#define S6_PWM        6
#define S7_BTN        7
#define S8_SSEG       8
#define S9_SPI        9
#define S10_I2C      10
#define S11_PS2      11
#define S12_DDFS     12
#define S13_ADSR     13
```

```
// video module definition
#define V0_SYNC      0
#define V1_MOUSE     1
#define V2_OSD       2
#define V3_GHOST     3
#define V4_USER4     4
#define V5_USER5     5
#define V6_GRAY      6
#define V7_BAR       7

// video frame buffer
#define FRAME_OFFSET 0x00c00000
#define FRAME_BASE  BRIDGE_BASE+FRAME_OFFSET

// #endif    // _NEXYS4
/*****************************************************************/

#ifdef __cplusplus
} // extern "C"
#endif

#endif  // _CHU_IO_MAP_INCLUDED
```

6.3.2 Chu_io_rw.h

Proporciona macros de lectura y escritura de registro de entradas y salida, es decir, establece unas funciones que simplemente con llamarlas para hacer uso de ellas, en este archivo esta programado lo que tienen que hacer. Por ejemplo, calcular la dirección base del espacio de memoria que se necesite, leer o escribir en un registro de E/S, etc. El código de programación de este archivo software es el que sigue:

```
#ifndef _CHU_IO_RW_H_INCLUDED
#define _CHU_IO_RW_H_INCLUDED

// library
#include <inttypes.h>    // to use unitN_t type
#ifdef __cplusplus
extern "C" {
#endif

/*****************************************************************
 * generic low-level read and write access
 *  - offset: 32-bit word offset relative to base
 *  - 4*offset used for byte address
 *  - must bypass data cache for I/O access
 *  - may be replaced with vendor provided macros
 *    (if _VENDOR_IO_ACCESS_USED is defined)
 *****************************************************************/
#ifndef _VENDOR_IO_ACCESS_USED

/**
 * read an io register.
 * @param base_addr base address of an io core
 * @param offset register word offset.
 * @return 32-bit data of the register
 * @note macro calculates the byte address of the register and then read
 */
#define io_read(base_addr, offset) \
   (*(volatile uint32_t *)((base_addr) + 4*(offset)))
```

```
/**
 * write an io register
 * @param base_addr base address of an io core
 * @param offset register word offset
 * @param data 32-bit data
 */
#define io_write(base_addr, offset, data) \
   (*(volatile uint32_t *)((base_addr) + 4*(offset)) = (data))

#endif  // _VENDOR_IO_ACCESS_USED
/**
 * calculate base address of a memory mapped io slot.
 * @param base base-address of FPro system.
 * @param slot designated io slot number.
 * @return base address of the slot
 * @note that there are 32 words per slot.
 */
#define get_slot_addr(base, slot) \


   ((uint32_t)((base) + (slot)*32*4))

/**
 * Calculate base address of a video system slot.
 * @param base base-address of FPro system.
 * @param sprite designated video sprite slot number.
 * @return base address of the sprite
 * @note there are 2^14 words per slot.
 *       0x008000000 is the memory space for video system
 */
#define get_sprite_addr(base, sprite) \


   ((uint32_t)((base) + 0x00800000 + (sprite)*16384*4))
#ifdef __cplusplus
} // extern "C"
#endif

#endif /* _CHU_IO_RW_H_INCLUDED */
```

A partir de ahora, los siguientes archivos software están divididos en dos partes para cada componente de los que se citan a continuación: una parte de definición de bloque o núcleo (extensión .*h*) y otra parte de implementación del bloque o núcleo (extensión .*cpp*).

6.3.3 `timer_core.h` y `timer_core.cpp`

El desarrollo de estos archivos software ha sido estudiado en la sección 5.2.6.3 donde también se encuentra el código de programación de cada archivo con su explicación.

6.3.4 `uart_core.h` y `uart _core.cpp`

El desarrollo de estos archivos software ha sido estudiado en la sección 5.4.4 donde también se encuentra el código de programación de cada archivo con su explicación.

6.3.5 `gpio_cores.h` y `gpio_cores.cpp`

El desarrollo de estos archivos software ha sido estudiado en la sección 5.2.6.2 donde también se encuentra el código de programación de cada archivo con su explicación.

6.3.6 `xadc_core.h` y `xadc _core.cpp`

El desarrollo de estos archivos software ha sido estudiado en la sección 5.5.3 donde también se encuentra el código de programación de cada archivo con su explicación.

6.3.7 `sseg_core.h` y `sseg _cores.cpp`

El desarrollo de estos archivos software ha sido estudiado en la sección 5.7.2.3 donde también se encuentra el código de programación de cada archivo con su explicación.

6.3.8 `chu_init.h` y `chu_init.cpp`

Estos archivos corresponden a las rutinas de utilidad, que son un conjunto de rutinas simples para acceder a la hora del sistema y poder comunicarse con una consola. La explicación completa de estos archivos software se encuentra en la sección 5.2.7 así como el código de programación que las define y las implementa.

6.3.9 `TFG_demo_final_prueba.cpp`

Este último archivo software es el que contempla todo lo anterior desarrollado en un solo archivo. El funcionamiento final de todo lo programado anteriormente, tanto la parte hardware como software, se establece programando este archivo.

En él se van haciendo llamadas a los bloques que se han ido viendo en esta sección y finalmente se ha programado la visión de una serie de salidas de cada bloque utilizado a gusto del creador de la demo.

A continuación, se va a explicar paso a paso el programa que realiza la demo una vez ejecutada, ya que se han ido probando cada uno de los distintos componentes uno por uno para comprobar la finalización del proyecto con éxito.

En primer lugar, se establece una pequeña función de código, llamada `timer_check`, que sirve de chequeo antes del programa principal, se puede decir que sería el punto 0, y su código es el siguiente:

```
void timer_check(GpoCore *led_p) {
int i;
for (i = 0; i < 5; i++) {
led_p->write(0xffff);
sleep_ms(500);
led_p->write(0x0000);
sleep_ms(500);
debug("timer check - (loop #)/now: ", i, now_ms());
}
}
```

En este código se programa el encendido y apagado de los 16 LEDs con periodos de 500 ms (milisegundos) durante 5 repeticiones.

A partir de aquí, mediante la activación de los LEDs se va indicando el número de la prueba (en modo binario) que se está realizando, funcionando de modo contínuo e infinito.

La segunda parte de programa que se explica es la descrita en el párrafo anterior, y es que cada vez que se visualiza la prueba de un componente, antes de iniciarse se indica el número que corresponde. El código de programación para la indicación del número de prueba (en binario) es el siguiente:

```
void show_test_id(int n, GpoCore *led_p) {
inti,ptn;
ptn=n;//1<<n;

   for (i = 0; i < 20; i++) {
led_p->write(ptn);
sleep_ms(30);
led_p->write(0);
sleep_ms(30);
}
}
```

La función anterior, se repite antes de la visualización de cada componente y le vamos dando el valor decimal deseado a *n* en orden, para que muestre con los leds ese número en formato binario antes de cada prueba.

La función 1 (valor de *n=1* en la función `show_test`), llamada `led_check`, hace uso de los 16 LEDs de la Nexys 4 DDR, de tal manera que los enciende durante 150 ms uno a uno de derecha a izquierda, repitiendo esto 2 veces. El código de programación de esta función es el siguiente:

```
void led_check(GpoCore *led_p, int n) {
int i, s;
for (s = 0; s < 2; s++) {
for (i = 0; i < n; i++) {
led_p->write(1, i);
sleep_ms(150);
led_p->write(0, i);
sleep_ms(150);
} //for i
} //for n
}
```

La función 2 (valor de *n=2* en la función `show_test`), llamada `sw_check`, hace una lectura del estado de los 16 switches o interruptores de la tarjeta y lo refleja encendiendo el LED correspondiente a cada switch. El encendido de los LEDs correspondientes lo realiza de forma intermitente cada 300 ms y durante 20 repeticiones. El código de esta función es:

```
void sw_check(GpoCore *led_p, GpiCore *sw_p) {
int i, s;

s = sw_p->read();
for (i = 0; i < 20; i++) {
led_p->write(s);
sleep_ms(300);
led_p->write(0);
sleep_ms(300);
}
}
```

La siguiente función (valor de *n=3* en la función `show_test`), corresponde al componente de la uart, mediante la función `uart_check`. Mediante esta función somos capaces de sacar sobre una consola (se explica más adelante) el texto programado, así como el texto que queramos teclear sobre un teclado de pc conectado al mismo pc del que se ejecuta el programa. En esta demo se sacan por pantalla de consola una serie de pruebas y con esta función se va indicando el número de prueba que es, como se muestra en el código siguiente:

```
void uart_check() {
static int loop = 0;
uart.disp("uart test #");
uart.disp(loop);
uart.disp("\n\r");
loop++;
}
```

Por medio del comando `uart.disp()`, se transmite información a la consola, donde se indica el número de prueba que es, `uart test #`.

La siguiente es una función de depuración de los valores de los switches y de los milisegundos guardados en las funciones anteriores, para que cuando se

ejecute el siguiente ciclo de todas las funciones estén actuaizados los cambios que haya podido haber por la manipulación del usuario. Se trata de la función 4 (valor de *n=4* en la función `show_test`), corresponde a la función llamada `debug` y su código es el siguiente:

```
debug("main - switch value / up time : ", sw.read(), now_ms());
```

La función número 5, (valor de *n=5* en la función `show_test`) llamada `adc_check`, hace uso del componente XADC (conversor analógico a digital), mediante el cual, en este proyecto, obtenemos la temperatura que tiene la propia tarjeta Nexys 4 DDR como se ha explicado en el apartado 5.5.3.

Esta función hace la lectura del voltaje por medio del Pmod JXADC, transformando esa tensión en temperatura como se ha explicado en la sección nombrada en el párrafo anterior.

Como parte de esta demo, se muestran en la consola que se explicará más adelante, las lecturas de tensión y temperatura de la FPGA así como el voltaje de cada uno de los 4 canales anañógicos conectados al Pmod JXADC, como se muestra en la Ilustración 40:

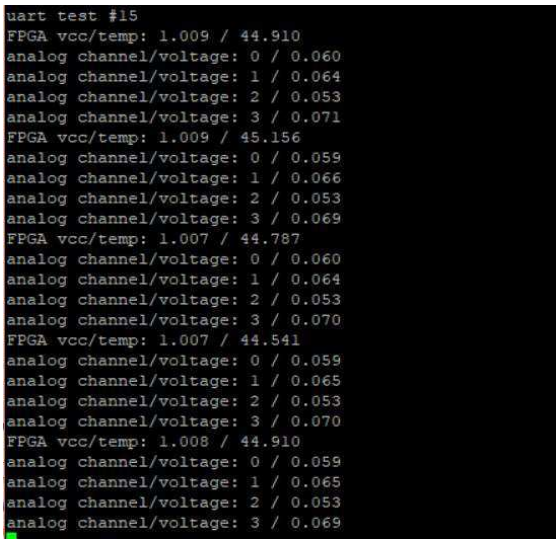


Ilustración 40: *Resultados de la demo en el terminal de transmisión serie*

El código utilizado para la función del componente XADC es el siguiente:

```
void adc_check(XadcCore *adc_p, GpoCore *led_p) {
double reading;
int n, i;
uint16_t raw;
for (i = 0; i < 5; i++) {
```

```
// display 12-bit channel 0 reading in LED
raw = adc_p->read_raw(0);
raw = raw >> 4;
led_p->write(raw);
// display on-chip sensor and 4 channels in console
uart.disp("FPGA vcc/temp: ");
reading = adc_p->read_fpga_vcc();
uart.disp(reading, 3);
uart.disp(" / ");
reading = adc_p->read_fpga_temp();
uart.disp(reading, 3);
uart.disp("\n\r");
for (n = 0; n < 4; n++) {
uart.disp("analog channel/voltage: ");
uart.disp(n);
uart.disp(" / ");
reading = adc_p->read_adc_in(n);
uart.disp(reading, 3);
uart.disp("\n\r");
} // end for
sleep_ms(200);
}
}
```

A continuación, se muestra la programación de la función 6 (valor de *n=6* en la función `show_test`), que mediante el bloque de modulación por anchura de pulso (PWM) se visualiza el funcionamiento de los dos LEDs de color RGB que contiene la placa.

La función de programación software que se utiliza es `pwm_3color_led_check`, que contiene el siguiente código:

```
void pwm_3color_led_check(PwmCore *pwm_p) {
int i, n;
double bright, duty;
const double P20 = 1.2589;  // P20=100^(1/20); i.e., P20^20=100
pwm_p->set_freq(50);
for (n = 0; n < 3; n++) {
bright = 1.0;
for (i = 0; i < 20; i++) {
bright = bright * P20;
duty = bright / 100.0;
pwm_p->set_duty(duty, n);
pwm_p->set_duty(duty, n + 3);
sleep_ms(100);
}
sleep_ms(300);
pwm_p->set_duty(0.0, n);
pwm_p->set_duty(0.0, n + 3);
}
}
```

El funcionamiento de estos LEDs de color trata de ver reflejado en ambos LEDs RGB los principales Red-Green-Blue (rojo, verde y azul) de forma alternada de modo que se va incrementando la intensidad del brillo de estos colores en 20 valores distintos desde un brillo casi ausente, hasta el brillo máximo posible.

El siguiente componente (valor de *n=7* en la función `show_test`) de la tarjeta del que se hace uso son los 5 botones que se encuentran en la parte inferior

derecha de la tarjeta Nexys 4 DDR, y que es posible su correcto funcionamiento gracias al uso del bloque de Antirrebote explicado en la sección 5.7.1.

El código programado es el siguiente y en el apartado 5.7.1.4 se explica su funcionamiento:

```
void debounce_check(DebounceCore *db_p, GpoCore *led_p) {
long start_time;
int btn_old, db_old, btn_new, db_new;
int b = 0;
int d = 0;
uint32_t ptn;

start_time = now_ms();
btn_old = db_p->read();
db_old = db_p->read_db();
do {
btn_new = db_p->read();
db_new = db_p->read_db();
if (btn_old != btn_new) {
b = b + 1;
btn_old = btn_new;
}
if (db_old != db_new) {
d = d + 1;
db_old = db_new;
}
ptn = d & 0x0000000f;
ptn = ptn | (b & 0x0000000f) << 4;
led_p->write(ptn);
} while ((now_ms() - start_time) < 5000);
}
```

Por último, se encuentra el componente del display de 7 segmentos compuesto por ocho dígitos, es la prueba número 8 (valor de *n=8* en la función `show_test`) y está basada en lo estudiado en la sección 5.7.2.

El código de programa es el siguiente, y a continuación se explica su funcionamiento de forma más detallada:

```
void sseg_check(SsegCore *sseg_p) {
int i, n, s, m;
uint8_t dp;

//turn off led
for (i = 0; i < 8; i++) {
sseg_p->write_1ptn(0xff, i);
}
//turn off all decimal points
sseg_p->set_dp(0x00);

for (s = 0; s < 3; s++) {
n = 0;
i = 0;
m = 200;

sseg_p->write_1ptn(sseg_p->h2s(n), i);
sleep_ms(m);
sseg_p->write_1ptn(sseg_p->h2s(n), i + 1);
sseg_p->write_1ptn(sseg_p->h2s(n + 1), i);
sleep_ms(m);
sseg_p->write_1ptn(sseg_p->h2s(n), i + 2);
sseg_p->write_1ptn(sseg_p->h2s(n + 1), i + 1);
sseg_p->write_1ptn(sseg_p->h2s(n + 2), i);
```

```
sleep_ms(m);
sseg_p->write_1ptn(sseg_p->h2s(n), i + 3);
sseg_p->write_1ptn(sseg_p->h2s(n + 1), i + 2);
sseg_p->write_1ptn(sseg_p->h2s(n + 2), i + 1);
sseg_p->write_1ptn(sseg_p->h2s(n + 16), i);
sleep_ms(m);
sseg_p->write_1ptn(sseg_p->h2s(n), i + 4);
sseg_p->write_1ptn(sseg_p->h2s(n + 1), i + 3);
sseg_p->write_1ptn(sseg_p->h2s(n + 2), i + 2);
sseg_p->write_1ptn(sseg_p->h2s(n + 16), i + 1);
sseg_p->write_1ptn(sseg_p->h2s(n + 3), i);
sleep_ms(m);
sseg_p->write_1ptn(sseg_p->h2s(n), i + 5);
sseg_p->write_1ptn(sseg_p->h2s(n + 1), i + 4);
sseg_p->write_1ptn(sseg_p->h2s(n + 2), i + 3);
sseg_p->write_1ptn(sseg_p->h2s(n + 16), i + 2);
sseg_p->write_1ptn(sseg_p->h2s(n + 3), i + 1);
sseg_p->write_1ptn(sseg_p->h2s(n + 4), i);
sleep_ms(m);
sseg_p->write_1ptn(sseg_p->h2s(n), i + 6);
sseg_p->write_1ptn(sseg_p->h2s(n + 1), i + 5);
sseg_p->write_1ptn(sseg_p->h2s(n + 2), i + 4);
sseg_p->write_1ptn(sseg_p->h2s(n + 16), i + 3);
sseg_p->write_1ptn(sseg_p->h2s(n + 3), i + 2);
sseg_p->write_1ptn(sseg_p->h2s(n + 4), i + 1);
sseg_p->write_1ptn(sseg_p->h2s(n + 5), i);
sleep_ms(m);
sseg_p->write_1ptn(sseg_p->h2s(n), i + 7);
sseg_p->write_1ptn(sseg_p->h2s(n + 1), i + 6);
sseg_p->write_1ptn(sseg_p->h2s(n + 2), i + 5);
sseg_p->write_1ptn(sseg_p->h2s(n + 16), i + 4);
sseg_p->write_1ptn(sseg_p->h2s(n + 3), i + 3);
sseg_p->write_1ptn(sseg_p->h2s(n + 4), i + 2);
sseg_p->write_1ptn(sseg_p->h2s(n + 5), i + 1);
sseg_p->write_1ptn(sseg_p->h2s(n + 6), i);
sleep_ms(m);
sseg_p->write_1ptn(sseg_p->h2s(n + 1), i + 7);
sseg_p->write_1ptn(sseg_p->h2s(n + 2), i + 6);
sseg_p->write_1ptn(sseg_p->h2s(n + 16), i + 5);
sseg_p->write_1ptn(sseg_p->h2s(n + 3), i + 4);
sseg_p->write_1ptn(sseg_p->h2s(n + 4), i + 3);
sseg_p->write_1ptn(sseg_p->h2s(n + 5), i + 2);
sseg_p->write_1ptn(sseg_p->h2s(n + 6), i + 1);
sseg_p->write_1ptn(sseg_p->h2s(n + 7), i);
sleep_ms(m);
sseg_p->write_1ptn(sseg_p->h2s(n + 2), i + 7);
sseg_p->write_1ptn(sseg_p->h2s(n + 16), i + 6);
sseg_p->write_1ptn(sseg_p->h2s(n + 3), i + 5);
sseg_p->write_1ptn(sseg_p->h2s(n + 4), i + 4);
sseg_p->write_1ptn(sseg_p->h2s(n + 5), i + 3);
sseg_p->write_1ptn(sseg_p->h2s(n + 6), i + 2);
sseg_p->write_1ptn(sseg_p->h2s(n + 7), i + 1);
sseg_p->write_1ptn(sseg_p->h2s(n + 8), i);
sleep_ms(m);
sseg_p->write_1ptn(sseg_p->h2s(n + 16), i + 7);
sseg_p->write_1ptn(sseg_p->h2s(n + 3), i + 6);
sseg_p->write_1ptn(sseg_p->h2s(n + 4), i + 5);
sseg_p->write_1ptn(sseg_p->h2s(n + 5), i + 4);
sseg_p->write_1ptn(sseg_p->h2s(n + 6), i + 3);
sseg_p->write_1ptn(sseg_p->h2s(n + 7), i + 2);
sseg_p->write_1ptn(sseg_p->h2s(n + 8), i + 1);
sseg_p->write_1ptn(sseg_p->h2s(n + 16), i);
sleep_ms(m);
sseg_p->write_1ptn(sseg_p->h2s(n + 3), i + 7);
sseg_p->write_1ptn(sseg_p->h2s(n + 4), i + 6);
sseg_p->write_1ptn(sseg_p->h2s(n + 5), i + 5);
sseg_p->write_1ptn(sseg_p->h2s(n + 6), i + 4);
sseg_p->write_1ptn(sseg_p->h2s(n + 7), i + 3);
sseg_p->write_1ptn(sseg_p->h2s(n + 8), i + 2);
sseg_p->write_1ptn(sseg_p->h2s(n + 16), i + 1);
sseg_p->write_1ptn(sseg_p->h2s(n + 9), i);
sleep_ms(m);
sseg_p->write_1ptn(sseg_p->h2s(n + 4), i + 7);
sseg_p->write_1ptn(sseg_p->h2s(n + 5), i + 6);
```

```
sseg_p->write_1ptn(sseg_p->h2s(n + 6), i + 5);
sseg_p->write_1ptn(sseg_p->h2s(n + 7), i + 4);
sseg_p->write_1ptn(sseg_p->h2s(n + 8), i + 3);
sseg_p->write_1ptn(sseg_p->h2s(n + 16), i + 2);
sseg_p->write_1ptn(sseg_p->h2s(n + 9), i + 1);
sseg_p->write_1ptn(sseg_p->h2s(n + 10), i);
sleep_ms(m);
sseg_p->write_1ptn(sseg_p->h2s(n + 5), i + 7);
sseg_p->write_1ptn(sseg_p->h2s(n + 6), i + 6);
sseg_p->write_1ptn(sseg_p->h2s(n + 7), i + 5);
sseg_p->write_1ptn(sseg_p->h2s(n + 8), i + 4);
sseg_p->write_1ptn(sseg_p->h2s(n + 16), i + 3);
sseg_p->write_1ptn(sseg_p->h2s(n + 9), i + 2);
sseg_p->write_1ptn(sseg_p->h2s(n + 10), i + 1);
sseg_p->write_1ptn(sseg_p->h2s(n + 11), i);
sleep_ms(m);
sseg_p->write_1ptn(sseg_p->h2s(n + 6), i + 7);
sseg_p->write_1ptn(sseg_p->h2s(n + 7), i + 6);
sseg_p->write_1ptn(sseg_p->h2s(n + 8), i + 5);
sseg_p->write_1ptn(sseg_p->h2s(n + 16), i + 4);
sseg_p->write_1ptn(sseg_p->h2s(n + 9), i + 3);
sseg_p->write_1ptn(sseg_p->h2s(n + 10), i + 2);
sseg_p->write_1ptn(sseg_p->h2s(n + 11), i + 1);
sseg_p->write_1ptn(sseg_p->h2s(n + 12), i);
sleep_ms(m);
sseg_p->write_1ptn(sseg_p->h2s(n + 7), i + 7);
sseg_p->write_1ptn(sseg_p->h2s(n + 8), i + 6);
sseg_p->write_1ptn(sseg_p->h2s(n + 16), i + 5);
sseg_p->write_1ptn(sseg_p->h2s(n + 9), i + 4);
sseg_p->write_1ptn(sseg_p->h2s(n + 10), i + 3);
sseg_p->write_1ptn(sseg_p->h2s(n + 11), i + 2);
sseg_p->write_1ptn(sseg_p->h2s(n + 12), i + 1);
sseg_p->write_1ptn(sseg_p->h2s(n + 13), i);
sleep_ms(m);
sseg_p->write_1ptn(sseg_p->h2s(n + 8), i + 7);
sseg_p->write_1ptn(sseg_p->h2s(n + 16), i + 6);
sseg_p->write_1ptn(sseg_p->h2s(n + 9), i + 5);
sseg_p->write_1ptn(sseg_p->h2s(n + 10), i + 4);
sseg_p->write_1ptn(sseg_p->h2s(n + 11), i + 3);
sseg_p->write_1ptn(sseg_p->h2s(n + 12), i + 2);
sseg_p->write_1ptn(sseg_p->h2s(n + 13), i + 1);
sseg_p->write_1ptn(sseg_p->h2s(n + 14), i);
sleep_ms(m);
sseg_p->write_1ptn(sseg_p->h2s(n + 16), i + 7);
sseg_p->write_1ptn(sseg_p->h2s(n + 9), i + 6);
sseg_p->write_1ptn(sseg_p->h2s(n + 10), i + 5);
sseg_p->write_1ptn(sseg_p->h2s(n + 11), i + 4);
sseg_p->write_1ptn(sseg_p->h2s(n + 12), i + 3);
sseg_p->write_1ptn(sseg_p->h2s(n + 13), i + 2);
sseg_p->write_1ptn(sseg_p->h2s(n + 14), i + 1);
sseg_p->write_1ptn(sseg_p->h2s(n + 15), i);
sleep_ms(m);
sseg_p->write_1ptn(sseg_p->h2s(n + 9), i + 7);
sseg_p->write_1ptn(sseg_p->h2s(n + 10), i + 6);
sseg_p->write_1ptn(sseg_p->h2s(n + 11), i + 5);
sseg_p->write_1ptn(sseg_p->h2s(n + 12), i + 4);
sseg_p->write_1ptn(sseg_p->h2s(n + 13), i + 3);
sseg_p->write_1ptn(sseg_p->h2s(n + 14), i + 2);
sseg_p->write_1ptn(sseg_p->h2s(n + 15), i + 1);
sseg_p->write_1ptn(sseg_p->h2s(n + 16), i);
sleep_ms(m);
sseg_p->write_1ptn(sseg_p->h2s(n + 10), i + 7);
sseg_p->write_1ptn(sseg_p->h2s(n + 11), i + 6);
sseg_p->write_1ptn(sseg_p->h2s(n + 12), i + 5);
sseg_p->write_1ptn(sseg_p->h2s(n + 13), i + 4);
sseg_p->write_1ptn(sseg_p->h2s(n + 14), i + 3);
sseg_p->write_1ptn(sseg_p->h2s(n + 15), i + 2);
sseg_p->write_1ptn(sseg_p->h2s(n + 16), i + 1);
sseg_p->write_1ptn(sseg_p->h2s(n + 16), i);
sleep_ms(m);
sseg_p->write_1ptn(sseg_p->h2s(n + 11), i + 7);
sseg_p->write_1ptn(sseg_p->h2s(n + 12), i + 6);
sseg_p->write_1ptn(sseg_p->h2s(n + 13), i + 5);
sseg_p->write_1ptn(sseg_p->h2s(n + 14), i + 4);
```

```
sseg_p->write_1ptn(sseg_p->h2s(n + 15), i + 3);
sseg_p->write_1ptn(sseg_p->h2s(n + 16), i + 2);
sseg_p->write_1ptn(sseg_p->h2s(n + 16), i + 1);
sseg_p->write_1ptn(sseg_p->h2s(n + 16), i);
sleep_ms(m);
sseg_p->write_1ptn(sseg_p->h2s(n + 12), i + 7);
sseg_p->write_1ptn(sseg_p->h2s(n + 13), i + 6);
sseg_p->write_1ptn(sseg_p->h2s(n + 14), i + 5);
sseg_p->write_1ptn(sseg_p->h2s(n + 15), i + 4);
sseg_p->write_1ptn(sseg_p->h2s(n + 16), i + 3);
sseg_p->write_1ptn(sseg_p->h2s(n + 16), i + 2);
sseg_p->write_1ptn(sseg_p->h2s(n + 16), i + 1);
sseg_p->write_1ptn(sseg_p->h2s(n + 16), i);
sleep_ms(m);
sseg_p->write_1ptn(sseg_p->h2s(n + 13), i + 7);
sseg_p->write_1ptn(sseg_p->h2s(n + 14), i + 6);
sseg_p->write_1ptn(sseg_p->h2s(n + 15), i + 5);
sseg_p->write_1ptn(sseg_p->h2s(n + 16), i + 4);
sseg_p->write_1ptn(sseg_p->h2s(n + 16), i + 3);
sseg_p->write_1ptn(sseg_p->h2s(n + 16), i 2);
sseg_p->write_1ptn(sseg_p->h2s(n + 16), i + 1);
sseg_p->write_1ptn(sseg_p->h2s(n + 16), i);
sleep_ms(m);
sseg_p->write_1ptn(sseg_p->h2s(n + 14), i + 7);
sseg_p->write_1ptn(sseg_p->h2s(n + 15), i + 6);
sseg_p->write_1ptn(sseg_p->h2s(n + 16), i + 5);
sseg_p->write_1ptn(sseg_p->h2s(n + 16), i + 4);
sseg_p->write_1ptn(sseg_p->h2s(n + 16), i + 3);
sseg_p->write_1ptn(sseg_p->h2s(n + 16), i + 2);
sseg_p->write_1ptn(sseg_p->h2s(n + 16), i + 1);
sseg_p->write_1ptn(sseg_p->h2s(n + 16), i);
sleep_ms(m);
sseg_p->write_1ptn(sseg_p->h2s(n + 15), i + 7);
sseg_p->write_1ptn(sseg_p->h2s(n + 16), i + 6);
sseg_p->write_1ptn(sseg_p->h2s(n + 16), i + 5);
sseg_p->write_1ptn(sseg_p->h2s(n + 16), i + 4);
sseg_p->write_1ptn(sseg_p->h2s(n + 16), i + 3);
sseg_p->write_1ptn(sseg_p->h2s(n + 16), i + 2);
sseg_p->write_1ptn(sseg_p->h2s(n + 16), i + 1);
sseg_p->write_1ptn(sseg_p->h2s(n + 16), i);
sleep_ms(m);
sseg_p->write_1ptn(sseg_p->h2s(n + 16), i + 7);
sseg_p->write_1ptn(sseg_p->h2s(n + 16), i + 6);
sseg_p->write_1ptn(sseg_p->h2s(n + 16), i + 5);
sseg_p->write_1ptn(sseg_p->h2s(n + 16), i + 4);
sseg_p->write_1ptn(sseg_p->h2s(n + 16), i + 3);
sseg_p->write_1ptn(sseg_p->h2s(n + 16), i + 2);
sseg_p->write_1ptn(sseg_p->h2s(n + 16), i + 1);
sseg_p->write_1ptn(sseg_p->h2s(n + 16), i);
sleep_ms(m);
} // for i
```

Para la realización de la demo, en cuanto a este componente se refiere, se ha realizado la visualización por medio de los ocho displays de 7 segmentos, durante las primeras 7 pruebas antes definidas y en forma de título, el mensaje “DEMO TFG”, y a modo de prueba más completa en la señal de LEDs número 8 en binario se visualiza de forma contínua durante tres ciclos y con movimiento de derecha a izquierda el siguiente mensaje personalizado “TFG HECTOR MANGADO” seguido de una demostración de funcionamiento de los puntos decimales que contiene cada display.

Finalmente, se adjunta a continuación el código de programa que hace

llamamiento a todas estas funciones software definidas y las ejecuta en el orden que se muestra, como se ha ido explicando con detalle:

```
int main() {
   //uint8_t id, ;
   timer_check(&led);
   while (1) {
   sleep_ms(1000);
   show_test_id(1, &led);
   led_check(&led, 16);
   sleep_ms(1000);
   show_test_id(2, &led);
   sw_check(&led, &sw);
   sleep_ms(1000);
   show_test_id(3, &led);
   uart_check();
   sleep_ms(1000);
   show_test_id(4, &led);
   debug("main - switch value / up time : ", sw.read(), now_ms());
   sleep_ms(1000);
   show_test_id(5, &led);
   adc_check(&adc, &led);
   sleep_ms(1000);
   show_test_id(6, &led);
   pwm_3color_led_check(&pwm);
   sleep_ms(1000);
   show_test_id(7, &led);
   debounce_check(&btn, &led);
   sleep_ms(1000);
   show_test_id(8, &led);
   sseg_check(&sseg);
   sleep_ms(1000);
   } //while
   sleep_ms(2000);
   timer_check(&led);
} //main
```

Se puede observar en el código que antes de cada una de las pruebas se hace llamada a la función `show_test` indicando el número de prueba que se realiza para ser mostrado por medio de los LEDs antes de ver la prueba y posteriormente hay unos comandos de espera para ver la ejecución del programa de una manera más cómoda.

6.4 Realización de pruebas

En cuanto a la realización de pruebas para comprobar el funcionamiento de la demo, se han podido hacer comprobaciones directamente manejando las distintas entradas de los componentes que proporciona la tarjeta Nexys 4 DDR y visualizando las propias salidas de la tarjeta, así como mediante la consola PuTTY se visualizan las pruebas realizadas mediante el bloque UART.

Las entradas manipulables de la tarjeta corresponden a los 16 switches, los 5 botones o pulsadores y el botón de reset, además existen entradas como la señal

rx (recepción de comunicación serie de la tarjeta), las señales de clk (reloj) y de adc_p y adc_n que son entradas sin acceso a ser modificadas por el usuario.

Como salidas el usuario puede visualizar en la tarjeta Nexys 4 DDR los 16 Leds, los ocho displays de 7 segmentos y los dos LEDs RGB de color; por otro lado, para visualizar todo lo que incluye al bloque UART, es necesario el uso de una consola de comunicación serie (PuTTY en este caso) para observar los resultados obtenidos de las pruebas realizadas como se ha mostrado en la Ilustración 40.

La configuración de la consola se realiza mediante el puerto serie del ordenador que ocupa la conexión de la tarjeta Nexys 4 DDR, con los parámetros de velocidad de 9600 baudios.

6.5 Conclusión

Después de realizar todo el estudio y desarrollo de aplicaciones programables en SoC de 32 bits sobre FPGA, teniendo una base de conocimientos en diseños electrónicos y programación informática y electrónica en lenguajes HDL y C/C++, no resulta muy complejo poder realizar proyectos personalizados a cualquier usuario que se lo proponga, pudiendo manejar una gran cantidad de componentes electrónicos de gran diversidad de opciones a modo lucrativo y sobre todo a nivel industrial.

La tarjeta Nexys 4 DDR dispone de una amplia gama de componentes que no han sido utilizados en la elaboración de este trabajo debido a la falta de tiempo en gran medida y también a que se han buscado los ejemplos de componentes más utilizados a nivel educativo en los estudios cursados en al Universidad de La Rioja.

7. Normas y referencias

En este apartado del documento, se indica la normativa aplicada y una relación de los documentos y programas utilizados:

7.1 Disposiciones legales y normas aplicadas

Por tratarse de un equipamiento que requiere alimentación eléctrica, hay que seguir la Directiva 2014/35/UE del parlamento europeo y del consejo de 26 de febrero de 2014 sobre la armonización de las legislaciones de los Estados miembros en materia de comercialización de material eléctrico destinado a utilizarse con determinados límites de tensión:

https://www.boe.es/doue/2014/096/L00357-00374.pdf . En este documento se expone las pautas que deben seguir los dispositivos que van a trabajar con tensiones bajas.

También hay que destacar que para que dicho producto se pueda comercializar en los países miembro de la Unión Europea debe superar los ensayos de compatibilidad electromagnética para obtener el sello CE. Toda la información necesaria sobre dicho procedimiento se puede ver en las decisiones adoptadas conjuntamente por el parlamento europeo y por el consejo: decisión no 768/2008/ce del parlamento europeo y del consejo de 9 de julio de 2008 sobre un marco común para la comercialización de los productos y por la que se deroga la Decisión 93/465/CEE del Consejo:

https://www.boe.es/doue/2008/218/L00082-00128.pdf .

7.2 Programas utilizados

- Vivado 2018.3 de Xilinx ISE WebPack
- Xilinx Software Development Kit (SDK) 2018.3
- PuTTY para control de dispositivos serie UART
- Microsoft Office 2013

7.3 Bibliografía

[1]. Libro en formato papel: FPGA PROTOTYPING BY VHDL EXAMPLES – Xilinx MicroBlaze MCS SoC 2ª edición. Pong P. Chu.

[2]. Academic.csuohio.edu (2019). Sitio web complementario del libro mencionado en el guion anterior https://academic.csuohio.edu/chu_p/rtl/fpga_mcs_vhdl.html. [Último acceso en julio de 2019]

[3]. Reference.digilentinc.com (2019). Manual de instrucciones de la tarjeta de Xilins, Nexys 4 DDR https://reference.digilentinc.com/reference/programmable-logic/nexys-4-ddr/reference-manual. [Último acceso en julio de 2019]

[4]. Xilinx.com, soporte y documentación (2019). Guía de inicio rápido y visión de la estructura de MicroBlaze https://www.xilinx.com/support/documentation/quick_start/microblaze-quick-start-guide.pdf. [Último acceso en julio de 2019]

[5]. Reference.digilentinc.com (2019). Esquemas de circuitería de Digilent de la tarjeta Nexys 4 DDR https://reference.digilentinc.com/_media/nexys4-ddr:nexys_4_ddr_sch.pdf. [Último acceso en julio de 2019]

[6]. Xilinx.com, soporte y documentación (2019). Hoja de datos de la tarjeta Nexys 4 DDR: https://www.xilinx.com/support/documentation/data_sheets/ds181_Artix_7_Data_Sheet.pdf [Último acceso en julio de 2019]

7.4 Otras referencias

[7]. Forums.xilinxs.com (2019). Foro de Xilinx consulta sobre el desarrollo de la aplicación SDK para manejar una UART: https://forums.xilinx.com/t5/Embedded-Development-Tools/UART-SDK-application/td-p/572300. [Último acceso en junio de 2019]

[8]. Forums.xilinxs.com (2019). Foro de Xilinx consulta sobre el funcionamiento del compilador en SDK: https://forums.xilinx.com/t5/Embedded-

Development-Tools/SDK-compiler-option/td-p/565242. [Último acceso en julio de 2019]

[9]. Forums.xilinxs.com (2019). Foro de Xilinx consulta sobre la compilación de bibliotecas IP de Xilinx: https://forums.xilinx.com/t5/Simulation-and-Verification/Compiling-Xilinx-IP-Libraries/td-p/890482. [Último acceso en julio de 2019]

[10]. Academic.csuohio.edu (2019). Tutorial actualizado de desarrollo de sistemas FPro: https://academic.csuohio.edu/chu_p/rtl/fpga_mcs_vhdl_book/appendix_revised.pdf. [Último acceso en julio de 2019]

[11]. Arantxa.ii.uam.es (2019). Tutorial de Xilinx MicroBlaze: http://arantxa.ii.uam.es/~ivan/microblaze-jcra04.pdf. [Último acceso en julio de 2019]

[12]. Xilinx.com, productos (2019). Estructura de Micro Blaze MCS: https://www.xilinx.com/products/design-tools/mb-mcs.html [Último acceso en julio de 2019]

[13]. Xilinx.com, productos (2019). Página de productos de Xilinx donde se encuentra Micro Blaze MCS: https://www.xilinx.com/products/design-tools/microblaze.html [Último acceso en julio de 2019]

[14]. Anysilicon.com (2019). Interfaz AXI, definición disponible en: https://anysilicon.com/understanding-axi-protocol-quick-introduction/ [Último acceso en julio de 2019]

[15]. Wikipedia.es (2019). GPIO, definición disponible en: https://es.wikipedia.org/wiki/GPIO [Último acceso en julio de 2019]

[16]. Wikipedia.es (2019). RISC, definición disponible en: https://es.wikipedia.org/wiki/Reduced_instruction_set_computing [Último acceso en julio de 2019]

[17]. Wikipedia.es (2019). ARM, definición disponible en: https://es.wikipedia.org/wiki/Arquitectura_ARM [Último acceso en julio de 2019]

8. Definiciones y abreviaturas

(1). FPGA

FPGA (del inglés field-programmable gate array traducido como matriz de puertas programables), es un dispositivo programable que contiene bloques de lógica que estan interconexionados pudiendo ser configurados en el momento, mediante un lenguaje de descripción especializado.

(2). IP core

Es un módulo de propiedad intelectual (IP), se trata de un bloque lógico que se utiliza en la fabricación de una FPGA como bloque periférico, es decir, cada uno de los periféricos usados o añadidos en una FPGA.

(3). Micro Blaze MCS

El módulo MicroBlaze MCS es un sistema de procesador altamente integrado destinado a aplicaciones de controlador y se entrega como un sistema preconfigurado en 3 etapas, que incluye el procesador de software RISC de 32 bits MicroBlaze. El desarrollo de software para MicroBlaze MCS se maneja a través del Kit de diseño de software (SDK). Compuesto por 3 etapas: acceso a memoria local, un módulo E/S acoplado y un conjunto estándar de periféricos con microcontrolador

(4). RISC

Del inglés Reduced Instruction Set Computer, en español Computador con Conjunto de Instrucciones Reducidas, es un tipo de diseño de CPU generalmente utilizado en microprocesadores o microcontroladores con las siguientes características fundamentales:

- Instrucciones de tamaño fijo y presentadas en un reducido número de formatos.
- Solo las instrucciones de carga y almacenamiento acceden a la memoria de datos.

Además estos procesadores suelen disponer de muchos registros de propósito general.

El objetivo de diseñar máquinas con esta arquitectura es posibilitar la segmentación y el paralelismo en la ejecución de instrucciones y reducir los accesos a memoria. Las máquinas RISC protagonizan la tendencia actual de construcción de microprocesadores. PowerPC,2 DEC Alpha, MIPS, ARM, SPARC son ejemplos de algunos de ellos.

(5). Slot

Un slot (también llamado slot de expansión o ranura de expansión) es un elemento de la placa base de un ordenador que permite conectar a ésta una tarjeta adaptadora adicional o de expansión, la cual suele realizar funciones de control de dispositivos periféricos adicionales, tales como monitores, impresoras o unidades de disco.

(6). E/S

Conjunto de entradas y salidas que contiene un módulo o bloque para la comunicación entre un sistema de procesamiento de información, tal como un ordenador, y el mundo exterior.

(7). SoC o Sistema Integrado

SoC (System-On-Chip) es un sistema que integra el procesador, la memoria y la red de conexión en un solo chip, lo que provoca una menor demanda de potencia del sistema. Un SoC típico consta de un núcleo de procesador de 32 bits y muchas funciones seleccionables que permiten diseñar un sistema íntegramente a medida, reduciendo espacios y consumos de potencia a lo estrictamente necesario. Estas funciones incluyen la memoria, interfaces de bus, módulos de E/S, decoders y soporte de red.

(8). MMIO

Se refiere a un subsistema de Mapeado de Memoria de entradas y salidas (I/O). Este subsistema provoca que el procesador no haga distinciones entre la memoria y los periféricos de E/S y utiliza las mismas instrucciones de lectura y escritura para acceder a los periféricos de E/S, por medio del mismo BUS de direcciones.

(9). UART

Son las siglas en inglés de Universal Asynchronous Receiver-Transmitter, en español: Transmisor-Receptor Asíncrono Universal, es el dispositivo que controla los puertos y dispositivos serie. Se encuentra integrado en la placa base o en la tarjeta adaptadora del dispositivo.

(10). XADC

Corresponde al módulo de Xilinx de un conversor analógico digital. Como su propio nombre indica está formado por el conjunto de ES, componentes y uniones para poder realizar esta conversión.

(11). PWM

La modulación por ancho de pulsos (también conocida como PWM, siglas en inglés de pulse-width modulation) de una señal o fuente de energía es una técnica en la que se modifica el ciclo de trabajo de una señal periódica (una senoidal o una cuadrada, por ejemplo), ya sea para transmitir información a través de un canal de comunicaciones o para controlar la cantidad de energía que se envía a una carga.

(12). VHDL

VHDL es un lenguaje de especificación definido por el IEEE (Institute of Electrical and Electronics Engineers) (ANSI/IEEE 1076-1993) utilizado para describir circuitos digitales y para la automatización de diseño electrónico . VHDL es acrónimo proveniente de la combinación de dos acrónimos: VHSIC (Very High Speed Integrated Circuit) y HDL (Hardware Description Language). Aunque puede ser usado de forma general para describir cualquier circuito digital se usa principalmente para programar PLD (Programable Logic Device - Dispositivo Lógico Programable), FPGA (Field Programmable Gate Array), ASIC y similares.

(13). C++

Es un lenguaje de programación diseñado con la intención de extender al lenguaje de programación C mecanismos que permiten la manipulación de objetos. En ese sentido, desde el punto de vista de los lenguajes orientados a objetos, el C++ es un lenguaje híbrido.

(14). Periférico

Son dispositivos electrónico físicos que permiten que la computadora interactúe con el mundo exterior. Son considerados tambien periféricos los sistemas que almacenan o archivan la información. ... Los periféricos permiten realizar operaciones de entrada/salida (E/S), de almacenamiento o de comunicación.

(15). Core (bloque/módulo/núcleo)

Un núcleo es parte de una CPU que recibe instrucciones y realiza cálculos o acciones, en base a esas instrucciones. Un conjunto de instrucciones puede permitir que un programa de software realice una función específica.

Los procesadores pueden tener un solo núcleo o múltiples núcleos. Un procesador con dos núcleos se denomina procesador de doble núcleo , cuatro núcleos es de cuatro núcleos , etc., hasta ocho núcleos. Cuantos más núcleos tiene un procesador, más conjuntos de instrucciones puede recibir y procesar al mismo tiempo, lo que hace que la computadora sea más rápida.

(16). Tecnología ASIC

Un circuito Integrado para aplicaciones específicas (o ASIC, por sus siglas en inglés) es un circuito integrado hecho a la medida para un uso en particular, en vez de ser concebido para propósitos de uso general.

(17). Acelerador hardware

En informática, la aceleración por hardware es el uso del hardware para realizar alguna función más rápido de lo que es posible usando software ejecutándose en una CPU de propósito general.

El hardware que realiza la aceleración, cuando se encuentra en una unidad separada de la CPU, es denominado acelerador por hardware, o a menudo más específicamente como un acelerador gráfico o unidad de coma flotante, etc. Estos términos, sin embargo, son antiguos y se han sustituido por términos menos descriptivos como "placa de vídeo" o "placa gráfica".

(18). Bitstream

Un flujo de bits FPGA es un archivo que contiene la información de programación para un FPGA. Un dispositivo Xilinx FPGA debe programarse utilizando un flujo de bits específico para que se comporte como una plataforma de hardware embebido. Este flujo de bits normalmente lo proporciona el diseñador de hardware que crea la plataforma integrada.

Programar un FPGA es el proceso de cargar un flujo de bits en el FPGA. Durante la fase de desarrollo, el dispositivo FPGA se programa mediante utilidades como Vivado® o mediante las opciones de menú en SDK. Estas herramientas transfieren el flujo de bits al FPGA a bordo. En el hardware de producción, el flujo de bits generalmente se coloca en una memoria no volátil, y el hardware está configurado para programar el FPGA cuando se enciende.

(19). VGA

VGA es la abreviatura de Video Graphics Array. Un puerto VGA utiliza un conector de 15 pines, generalmente está situada en la parte posterior o lateral de una computadora. VGA es la interfaz estándar que sirve para conectar un monitor o proyector a los equipos.

(20). Memoria RAM

La Memoria RAM es un chip o tarjeta que forma parte de un dispositivo electrónico como un ordenador o un teléfono que sirve para el almacenamiento de información o datos de acceso directo. RAM son una siglas en inglés que significan «Random Access Memory» al traducirlo al español obtenemos «Memoria de Acceso Aleatorio».

(21). Cache

Una caché es un componente de hardware o software que almacena datos para que las solicitudes futuras de esos datos se puedan atender con mayor rapidez; los datos almacenados en un caché pueden ser el resultado de un cálculo anterior o el duplicado de datos almacenados en otro lugar, generalmente, da velocidad de acceso más rápido.

(22). Digilent

Digilent Inc. es una empresa de productos de ingeniería electrónica que brinda servicios a estudiantes, universidades y OEM alrededor del mundo con herramientas de diseño educacional basado en la tecnología.

(23). AXI (Interfaz eXtensible Avanzado)

Es unainterfaz de interconexión punto a punto diseñada para sistemas de microcontroladores de alto rendimiento y alta velocidad. El protocolo AXI se basa en una interconexión punto a punto para evitar compartir el bus y, por lo tanto, permitir un mayor ancho de banda y una menor latencia. AXI es posiblemente la más popular de todas las interconexiones de interfaz AMBA.

La esencia del protocolo AXI es que proporciona un marco para la forma en que diferentes bloques dentro de cada chip se comunican entre sí. Ofrece un procedimiento antes de que se transmita algo, para que la comunicación sea clara e ininterrumpida. De esa manera, diferentes componentes pueden comunicarse entre sí sin pisar uno del otro.

(24). ARM

Es una arquitectura RISC de 32 bits y, con la llegada de su versión V8-A, también de 64 Bits, desarrollada por ARM Holdings. Se llamó Advanced RISC Machine, y anteriormente Acorn RISC Machine. La arquitectura ARM es el conjunto de instrucciones de 32 y 64 bits más ampliamente utilizado en unidades producidas. Los procesadores ARM requieren una cantidad menor de transistores que los procesadores x86 CISC, típicos en la mayoría de ordenadores personales. Este enfoque de diseño nos lleva, por tanto, a una reducción de los costes, calor y energía.

(25). Memoria FIFO

FIFO es el acrónimo inglés de First In, First Out (primero en entrar, primero en salir). Es un método utilizado en estructuras de datos, contabilidad de costes y teoría de colas.

FIFO se utiliza en estructuras de datos para organizar colas de datos. La implementación puede efectuarse con ayuda de arrays o vectores, o bien mediante el uso de punteros y asignación dinámica de memoria .Si se

implementa mediante vectores el número máximo de elementos que puede almacenar está limitado al que se haya establecido en el código del programa antes de la compilación (cola estática) o durante su ejecución.

(26). PLL (Phase Locked Loop)

Es un dispositivo electrónico de amplia aplicación en sistemas de comunicación. Básicamente consiste en un sistema de lazo cerrado capaz de enclavar (o sincronizar) la fase de un oscilador controlado por voltaje (VCO), con la fase de una señal de entrada.

(27). GPIO

GPIO (General Purpose Input/Output, Entrada/Salida de Propósito General) es un pin genérico en un chip, cuyo comportamiento (incluyendo si es un pin de entrada o salida) se puede controlar (programar) por el usuario en tiempo de ejecución.

Los pines GPIO no tienen ningún propósito especial definido, y no se utilizan de forma predeterminada. La idea es que a veces, para el diseño de un sistema completo que utiliza el chip podría ser útil contar con un puñado de líneas digitales de control adicionales, y tenerlas a disposición ahorra el tiempo de tener que organizar circuitos adicionales para proporcionarlos.

(28). Sistema Vanilla

Con este término extraído del inglés (en español «vainilla») se suelen conocer los programas o sistemas operativos que no han sufrido cambios, adaptaciones o actualizaciones con respecto a su versión original. El término alude al sabor original del helado, antes de que se añadiesen sabores más sofisticados.

(29). Offset

un offset dentro de un array u otra estructura de datos es un entero que indica la distancia (desplazamiento) desde el inicio del objeto hasta un punto o elemento dado, presumiblemente dentro del mismo objeto. El concepto de distancia es solamente válido si todos los elementos del objeto son del mismo tamaño (típicamente dados en bytes o palabras).

(30). MSB

En electrónica, el bit más significativo, most significant bit (MSB), en sus siglas en inglés, es el bit, que de acuerdo a su posición, tiene el mayor valor. En ocasiones, se hace referencia al MSB como el bit del extremo izquierdo.

(31). LSB

El bit menos significativo (LSB o Least Significant Bit, en sus siglas en inglés) es la posición de bit en un número binario que tiene el menor valor (el situado más a la derecha). ... El LSB, escrito en mayúsculas, puede también significar "Byte Menos Significativo" (Least Significant Byte).

(32). Puntero

un puntero es un objeto del lenguaje de programación, cuyo valor se refiere a (o "apunta a") otro valor almacenado en otra parte de la memoria del ordenador utilizando su dirección. Un puntero referencia a una ubicación en memoria, y a la obtención del valor almacenado en esa ubicación se la conoce como desreferenciación del puntero.

(33). LED

La tecnología conocida como LED (por sus siglas en inglés, Light Emitting Diode, que en español significa Diodo Emisor de Luz) también conocida como Diodo Luminoso consiste básicamente en un material semiconductor que es capaz de emitir una radiación electromagnética en forma de Luz.

(34). Protocolo handshake

Es utilizado en tecnologías informáticas, telecomunicaciones, y otras conexiones para establecer automáticamente una negociación entre pares que establece de forma dinámica los parámetros de un canal de comunicación entre ellos antes de que comience la comunicación normal por el canal. De ello se desprende la creación física del canal y precede a la transferencia de información normal.

(35). ASMD

El método de la máquina de estado algorítmico con Datapath (ASMD) es un método para diseñar máquinas de estado finito . Se utiliza para representar

diagramas de circuitos integrados digitales . El diagrama ASMD es como un diagrama de estado pero menos formal y, por lo tanto, más fácil de entender. Un gráfico ASMD es un método para describir las operaciones secuenciales de un sistema digital.

(36). FSMD

Una máquina de estados finitos con Datapath (FSMD) es una abstracción matemática que a veces se usa para diseñar programas de lógica digital o de computadora.

Un FSMD es un sistema digital compuesto por una máquina de estado finito , que controla el flujo del programa , y una ruta de datos , que realiza operaciones de procesamiento de datos. Son esencialmente programas secuenciales en los que las declaraciones se han programado en estados, lo que resulta en diagramas de estado más complejos.

Un programa se convierte en un diagrama de estado complejo en el que los estados y los arcos pueden métodos incluir expresiones aritméticas , y esas expresiones pueden usar entradas y salidas externas, así como variables.

Las FSM no utilizan variables ni operaciones / condiciones aritméticas, por lo que las FSMD son más poderosas que las FSM.

(37). Registro Flip-Flop

Los registros de desplazamiento son circuitos secuenciales formados por biestables o flip-flops generalmente de tipo D conectados en serie y una circuiteria adicional que controlará la manera de cargar y acceder a los datos que se almacenan.

(38). Sobrecarga de métodos

La sobrecarga de métodos es la creación de varios métodos con el mismo nombre, pero con diferentes firmas y definiciones. Se utiliza el número y tipo de argumentos para seleccionar qué definición de método ejecutar.

Esto ofrece flexibilidad a la hora de llamar a un método usando un número diferente de parámetros sin por ello, tener que replicar código.

(39). SPS (Samples Per Second)

Se trata de una unidad de tiempo. En el procesamiento de señales , el muestreo es la reducción de una señal de tiempo continuo a una señal de tiempo discreto. Un ejemplo común es la conversión de una onda de sonido (una señal continua) a una secuencia de muestras (una señal de tiempo discreto).

(40). Buffer

es un espacio de memoria, en el que se almacenan datos de manera temporal, normalmente para un único uso (generalmente utilizan un sistema de cola FIFO); su principal uso es para evitar que el programa o recurso que los requiere, ya sea hardware o software, se quede sin datos durante una transferencia (entrada/salida) de datos irregular o por la velocidad del proceso.

(41). RGB LED

LED RGB significa LED rojo, azul y verde. Los productos LED RGB combinan estos tres colores para producir más de 16 millones de tonos de luz. Pero no todos los colores son posibles. Algunos se encuentran "fuera" del triángulo formado por los LED RGB. Además, los colores pigmento, como el marrón o el rosa, son difíciles o imposibles de lograr.

ANEXOS

Índice

Índice de Listados

1. Código de programación

1.1 Código Hardware

En este apartado se adjunta todo el código de tipo hardware generado para la realización de este proyecto:

Listado 2.1. Definición de constantes y ranuras (chu_io_map.h)

```
#ifndef _CHU_IO_MAP_INCLUDED
#define _CHU_IO_MAP_INCLUDED

#ifdef __cplusplus
extern "C" {
#endif

//////////////////////////////////////
   ////////////////////////////*/
// #ifdef _NEXYS4

// system clock rate in MHz; used for
   timer and uart
#define SYS_CLK_FREQ 100

//io base address for microBlaze MCS
#define BRIDGE_BASE 0xc0000000

// slot module definition
// format: Slot#_ModuleType_Name
#define S0_SYS_TIMER  0
#define S1_UART1      1
#define S2_LED        2
#define S3_SW         3
#define S4_USER       4
#define S5_XDAC       5
#define S6_PWM        6
#define S7_BTN        7
#define S8_SSEG       8
#define S9_SPI        9
#define S10_I2C      10
#define S11_PS2      11
#define S12_DDFS     12
#define S13_ADSR     13

// video module definition
#define V0_SYNC       0
#define V1_MOUSE      1
#define V2_OSD        2
#define V3_GHOST      3
#define V4_USER4      4
#define V5_USER5      5
#define V6_GRAY       6
#define V7_BAR        7

// video frame buffer
#define FRAME_OFFSET 0x00c00000
#define FRAME_BASE
   BRIDGE_BASE+FRAME_OFFSET

// #endif    // _NEXYS4
/////////////////////////////////////
   ////////////////////////////*/

#ifdef __cplusplus
} // extern "C"
#endif

#endif  // _CHU_IO_MAP_INCLUDED
```

Listado 2.2. Macros de E/S (chu_io_rw.h)

```
#ifndef _CHU_IO_RW_H_INCLUDED
#define _CHU_IO_RW_H_INCLUDED

// library
#include <inttypes.h>    // to use
   unitN_t type
#ifdef __cplusplus
extern "C" {
#endif

#define io_read(base_addr, offset) \
   (*(volatile uint32_t *)((base_addr) +
   4*(offset)))

#define io_write(base_addr, offset,
   data) \
   (*(volatile uint32_t *)((base_addr) +
   4*(offset)) = (data))

#define get_slot_addr(base, slot) \
      ((uint32_t)((base) + (slot)*32*4))

#define get_sprite_addr(base, sprite) \
   ((uint32_t)((base) + 0x00800000 +
   (sprite)*16384*4))

#ifdef __cplusplus
} // extern "C"
#endif

#endif /* _CHU_IO_RW_H_INCLUDED */
```

Listado 2.3. Definición de la clase GpoCore (gpio_core.h)

```
class GpoCore {
/* register map */
   enum {
      DATA_REG = 0 //data register
   };
public:
   GpoCore(uint32_t core_base_addr);
```

```
   //constructor
   ~GpoCore();
   //destructor; not used
/* methods */
   void write(uint32_t data);
   //write a 32-bit word
   void write(int bit_value, int
   bit_pos); //write 1 bit
private:
   uint32_t base_addr;
   uint32_t wr_data;
   // same as GPO core data reg
};
```

Listado 2.4. Implementación de la clase GpoCore (gpio_core.cpp)

```
GpoCore::GpoCore(uint32_t
   core_base_addr) {
   base_addr = core_base_addr;
   wr_data = 0;
}

GpoCore::~GpoCore() {}

void GpoCore::write(uint32_t data) {
   wr_data = data;
   io_write(base_addr, DATA_REG,
   wr_data);
}

void GpoCore::write(int bit_value,
   int bit_pos) {
   bit_write(wr_data, bit_pos,
   bit_value);
   io_write(base_addr, DATA_REG,
   wr_data);
}
```

Listado 2.5. Definición de la clase GpiCore (gpio_core.h)

```
class GpiCore {
   /* register map */
   enum {
      DATA_REG = 0 /**< input data
   register */
   };
public:
   GpiCore(uint32_t core_base_addr);
   //constructor
   ~GpiCore();
   //destructor; not used
   /* methods */
   uint32_t read();
   //read a 32-bit word

   int read(int bit_pos);
   //read 1 bit
private:
   uint32_t base_addr;};
```

Listado 2.6. Implementación de la clase GpiCore (gpio_core.cpp)

```
GpiCore::GpiCore(uint32_t
   core_base_addr) {
   base_addr = core_base_addr;
}
GpiCore::~GpiCore() {
}

uint32_t GpiCore::read() {
   return (io_read(base_addr,
   DATA_REG));
}

int GpiCore::read(int bit_pos) {
   uint32_t rd_data =
   io_read(base_addr, DATA_REG);
   return ((int) bit_read(rd_data,
   bit_pos));
   }
```

Listado 2.7. Definición de la clase TimerCore (timer_core.h)

```
class TimerCore {
   */ register map */
   enum {
      COUNTER_LOWER_REG = 0, /**<
   lower 32 bits of counter */
      COUNTER_UPPER_REG = 1, /**<
   upper 16 bits of counter */
      CTRL_REG = 2           /**<
   control register */
   };
   /* masks */
   enum {
      GO_FIELD = 0x00000001, /**<
   bit 0 of ctrl_reg; enable bit  */
      CLR_FIELD = 0x00000002 /**<
   bit 1 of ctrl_reg; clear bit */
   };
public:
   TimerCore(uint32_t
   core_base_addr);    //constructor
   ~TimerCore();
   //destructor; not used
   /* methods */
   void pause();
   //pause counter
   void go();
   //resume counter
   void clear();
   //clear the counter to 0
   uint64_t read_tick();
   //retrieve # clocks elapsed
   uint64_t read_time();
   //read time elapsed (in
   microsecond)
   void sleep(uint64_t us);
   //idle for us microseconds
```

```
private:
   uint32_t base_addr;
   uint32_t ctrl;
   // current state of control
   register
   };
```

Listado 2.8. Implementación de la clase TimerCore (timer_core.cpp)

```
TimerCore::TimerCore(uint32_t
   core_base_addr) {
   base_addr = core_base_addr;
   ctrl = 0x01;
   io_write(base_addr, CTRL_REG,
   ctrl);  // enable the timer
}

TimerCore::~TimerCore() {}

void TimerCore::pause() {
   // reset enable bit to 0
   ctrl = ctrl & ~GO_FIELD;
   io_write(base_addr, CTRL_REG,
   ctrl);
}

void TimerCore::go() {
   // set enable bit to 1
   ctrl = ctrl | GO_FIELD;
   io_write(base_addr, CTRL_REG,
   ctrl);
}

void TimerCore::clear() {
   uint32_t wdata;

   // write clear_bit to generate a
   1-clock pulse
   // clear bit does not affect ctrl
   wdata = ctrl | CLR_FIELD;
   io_write(base_addr, CTRL_REG,
   wdata);
}

uint64_t TimerCore::read_tick() {
   uint64_t upper, lower;

   lower = (uint64_t)
   io_read(base_addr,
   COUNTER_LOWER_REG);
   upper = (uint64_t)
   io_read(base_addr,
   COUNTER_UPPER_REG);
   return ((upper << 32) | lower);
}

uint64_t TimerCore::read_time() {
   // elapsed time in microsecond
   (SYS_CLK_FREQ in MHz)
   return (read_tick() /
   SYS_CLK_FREQ);
}

void TimerCore::sleep(uint64_t us) {
   uint64_t start_time, now;

   start_time = read_time();
   // busy waiting
   do {
      now = read_time();
   } while ((now - start_time) <
   us);
   }
```

Listado 2.9. Declaraciones y macros de las rutinas de utilidad FPro (chu_init.h)

```
// library
#include "chu_io_rw.h"
#include "chu_io_map.h"
#include "timer_core.h"
#include "uart_core.h"

//  make uart visible by other code
extern UartCore uart;

// define timer and uart slots
#define TIMER_SLOT 0
#define UART_SLOT 1

//timing functions
unsigned long now_us();
unsigned long now_ms();
void sleep_us(unsigned long int t);
void sleep_ms(unsigned long int t);

//define debug function
void debug_off();
void debug_on(const char *str, int
   n1, int n2);

#ifndef _DEBUG
#define debug(str, n1, n2)
   debug_off()
#endif // not _DEBUG

#ifdef _DEBUG
#define debug(str, n1, n2)
   debug_on((str), (n1), (n2))
#endif // not _DEBUG

//low-level bit-manipulation macros
#define bit_set(data, n) ((data) |=
   (1UL << (n)))
#define bit_clear(data, n) ((data)
   &= ~(1UL << (n)))
#define bit_toggle(data, n) ((data)
   ^= (1UL << (n)))
#define bit_read(data, n) (((data)
   >> (n)) & 0x01)
#define bit_write(data, n, bitvalue)
   (bitvalue ? bit_set((data), n) :
   bit_clear((data), n))
#define bit(n) (1UL << (n))

#endif  // _CHU_INIT_H_INCLUDED
```

Listado 2.10. Implementación rutinas de utilidad FPro (chu_init.cpp)

```
TimerCore
   _sys_timer(get_slot_addr(BRIDGE_B
   ASE, TIMER_SLOT));
UartCore
   uart(get_slot_addr(BRIDGE_BASE,
   UART_SLOT));

// current system time in
   microsecond
unsigned long now_us() {
   return ((unsigned long)
   _sys_timer.read_time());
}

// current system time in ms
unsigned long now_ms() {
   return ((unsigned long)
   _sys_timer.read_time() / 1000);
}

// idle for t microseconds
void sleep_us(unsigned long int t) {
   _sys_timer.sleep(uint64_t(t));
}

// idle for t ms
void sleep_ms(unsigned long int t) {
   _sys_timer.sleep(uint64_t(1000 *
   t));
}

// uart print a 1-line message: msg
   + 2 numbers in dec/hex format
void debug_on(const char *str, int
   n1, int n2) {
   uart.disp("debug: ");
   uart.disp(str);
   uart.disp(n1);
   uart.disp("(0x");
   uart.disp(n1, 16);
   uart.disp(") / ");
   uart.disp(n2);
   uart.disp("(0x");
   uart.disp(n2, 16);
   uart.disp(") \n\r");
}

void debug_off() {
}
```

Listado 2.11. Programa de prueba Vanilla FPro (main_vanilla_test.cpp)

```
//#define _DEBUG
#include "chu_init.h"
#include "gpio_cores.h"

void timer_check(GpoCore *led_p) {
   int i;

   for (i = 0; i < 5; i++) {
      led_p->write(0xffff);
      sleep_ms(500);
      led_p->write(0x0000);
      sleep_ms(500);
      debug("timer check - (loop
   #)/now: ", i, now_ms());
   }
}

void led_check(GpoCore *led_p, int
   n) {
   int i;

   for (i = 0; i < n; i++) {
      led_p->write(1, i);
      sleep_ms(200);
      led_p->write(0, i);
      sleep_ms(200);
   }
}

void sw_check(GpoCore *led_p,
   GpiCore *sw_p) {
   int i, s;

   s = sw_p->read();
   for (i = 0; i < 30; i++) {
      led_p->write(s);
      sleep_ms(50);
      led_p->write(0);
      sleep_ms(50);
   }
}

void uart_check() {
   static int loop = 0;

   uart.disp("uart test #");
   uart.disp(loop);
   uart.disp("\n\r");
   loop++;
}

// instantiate switch, led
GpoCore
   led(get_slot_addr(BRIDGE_BASE,
   S2_LED));
GpiCore
   sw(get_slot_addr(BRIDGE_BASE,
   S3_SW));

int main() {

   while (1) {
      timer_check(&led);
      led_check(&led, 16);
      sw_check(&led, &sw);
      uart_check();
      debug("main - switch value /
   up time : ", sw.read(),
```

```
   now_ms());
   } //while
} //main
```

Listado 3.1. Núcleo GPO

```
library ieee;
use ieee.std_logic_1164.all;
entity chu_gpo is
   generic(W : integer := 8);  --
   width of output port
   port(
      clk     : in  std_logic;
      reset   : in  std_logic;
      -- slot interface
      cs      : in  std_logic;
      write   : in  std_logic;
      read    : in  std_logic;
      addr    : in
   std_logic_vector(4 downto 0);
      rd_data : out
   std_logic_vector(31 downto 0);
      wr_data : in
   std_logic_vector(31 downto 0);
      -- external signal
      dout    : out
   std_logic_vector(W - 1 downto 0)
   );
end chu_gpo;

architecture arch of chu_gpo is
   signal buf_reg :
   std_logic_vector(W - 1 downto 0);
   signal wr_en   : std_logic;
begin
   -- output buffer register
   process(clk, reset)
   begin
      if (reset = '1') then
         buf_reg <= (others => '0');
      elsif (clk'event and clk =
   '1') then
         if wr_en = '1' then
            buf_reg <= wr_data(W - 1
   downto 0);
         end if;
      end if;
   end process;
   -- decoding logic
   wr_en   <= '1' when write = '1'
   and cs = '1' else '0';
   -- slot read interface
   rd_data <= (others => '0');
   -- not used
   -- external output
   dout    <= buf_reg;
  end arch
```

Listado 3.2. Núcleo GPI

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity chu_gpi is
   generic(W : integer := 8);  --
   width of input port
   port(
      clk     : in  std_logic;
      reset   : in  std_logic;
      -- slot interface
      cs      : in  std_logic;
      write   : in  std_logic;
      read    : in  std_logic;
      addr    : in
   std_logic_vector(4 downto 0);
      rd_data : out
   std_logic_vector(31 downto 0);
      wr_data : in
   std_logic_vector(31 downto 0);
      -- external signal
      din     : in
   std_logic_vector(W-1 downto 0)
   );
end chu_gpi;

architecture arch of chu_gpi is
   signal rd_data_reg :
   std_logic_vector(W-1 downto 0);
begin
   -- input register
   process(clk, reset)
   begin
      if reset = '1' then
         rd_data_reg <= (others =>
   '0');
      elsif (clk'event and clk =
   '1') then
         rd_data_reg <= din;
      end if;
   end process;
   -- slot read interface
   rd_data(W-1 downto 0) <=
   rd_data_reg;
   rd_data(31 downto W)  <= (others
   => '0');
  end arch;
```

Listado 3.3. Núcleo Temporizador

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity chu_timer is
   port(
      clk     : in  std_logic;

      reset   : in  std_logic;
      -- slot interface
      cs      : in  std_logic;
      write   : in  std_logic;
      read    : in  std_logic;
      addr    : in
   std_logic_vector(4 downto 0);
      rd_data : out
   std_logic_vector(31 downto 0);
      wr_data : in
   std_logic_vector(31 downto 0)
   );
```

```
end chu_timer;

architecture arch of chu_timer is
   signal count_reg  : unsigned(47
   downto 0);
   signal count_next : unsigned(47
   downto 0);
   signal ctrl_reg   : std_logic;
   signal wr_en      : std_logic;
   signal clear, go  : std_logic;
begin
   --
   ********************************
   ********************************
   -- counter
   --
   ********************************
   ********************************
   -- register
   process(clk, reset)
   begin
      if reset = '1' then
         count_reg <= (others =>
   '0');
      elsif (clk'event and clk =
   '1') then
         count_reg <= count_next;
      end if;
   end process;
   -- next-state logic
   count_next <= (others => '0')
   when clear = '1' else
                 count_reg + 1
   when
go = '1' else
                 count_reg;

   --
   ********************************
   ********************************
   -- wrapping circuit
   --
   ********************************
   ********************************
  -- ctrl register
   process(clk, reset)
   begin
      if reset = '1' then
         ctrl_reg <= '0';
      elsif (clk'event and clk =
   '1') then
         if wr_en = '1' then
            ctrl_reg <= wr_data(0);
         end if;
      end if;
   end process;
   -- decoding logic
   wr_en <=
      '1' when write='1' and cs='1'
   and addr(1 downto 0)="10" else
   '0';
   clear <= '1' when wr_en='1' and
   wr_data(1)='1' else '0';
   go    <= ctrl_reg;
   -- slot read multiplexing
   rd_data <=
      std_logic_vector(count_reg(31
   downto 0)) when addr(0)='0' else
      x"0000" &
   std_logic_vector(count_reg(47
   downto 32));
  end arch;
```

Listado 3.4. Tipos de datos y declaraciones constantes en el paquete chu_io_map

```
library ieee;
use ieee.std_logic_1164.all;

package chu_io_map is
   --
   ********************************
   ********************************
   -- 2D data types
   --
   ********************************
   ********************************
   type slot_2d_data_type is array
   (63 downto 0) of
        std_logic_vector(31 downto
   0);
   type slot_2d_reg_type is array
   (63 downto 0) of
        std_logic_vector(4 downto
   0);
   type slot_2d_video_data_type is
   array (7 downto 0) of
        std_logic_vector(31 downto
   0);
   type slot_2d_video_reg_type is
   array (7 downto 0) of
        std_logic_vector(13 downto
   0);

   --
   ********************************
   ********************************
   -- Base address for the io_bridge
   --
   ********************************
   ********************************
   -- for xilinx MCS
   constant BRIDGE_BASE :
   std_logic_vector(31 downto 0) :=
   X"c7000000";

   --
   ********************************
   ********************************
   -- slot definition for the
   "sampler" MMIO subsystem
   -- avoid changing the first four
   slots
   --
   ********************************
   ********************************
   constant S0_SYS_TIMER : integer
```

```
:= 0;
constant S1_UART1     : integer
:= 1;
constant S2_LED       : integer
:= 2;
constant S3_SW        : integer
:= 3;
constant S4_USER      : integer
:= 4;
constant S5_XADC      : integer
:= 5;
constant S6_PWM       : integer
:= 6;
constant S7_BTN       : integer
:= 7;
constant S8_SSEG      : integer
:= 8;
constant S9_SPI       : integer
:= 9;
constant S10_I2C      : integer
:= 10;
constant S11_PS2      : integer
:= 11;
constant S12_DDFS     : integer
:= 12;
constant S13_ADSR     : integer
:= 13;

--
********************************
********************************
-- slot definition for the daisy
video subsystem
--
********************************
********************************
constant V0_SYNC  : integer := 0;
constant V1_MOUSE : integer := 1;
constant V2_OSD   : integer := 2;
constant V3_GHOST : integer := 3;
constant V4_USER4 : integer := 4;
constant V5_USER5 : integer := 5;
constant V6_GRAY  : integer := 6;
constant V7_BAR   : integer := 7;

--
********************************
********************************
-- addional slot definition for
the "dlx" MMIO subsystem
--
********************************
********************************
constant S14_USER1 : integer :=
14;
constant S15_USER2 : integer :=
15;
constant S16_TIMER1 : integer :=
16;
constant S17_TIMER2 : integer :=
17;
constant S18_UART2 : integer :=
18;
constant S19_UART3 : integer :=
19;
   constant S20_SPI1 : integer :=
   20;
   constant S21_SPI2 : integer :=
   21;
   constant S22_I2C1 : integer :=
   22;
   constant S23_I2C2 : integer :=
   23;
   constant S24_DDFS1 : integer :=
   24;
   constant S25_DDFS2 : integer :=
   25;
end chu_io_map;
```

Listado 3.5. Controlador MMIO

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use work.chu_io_map.all;
entity chu_mmio_controller is
   port(
      -- FPro bus
      mmio_cs             : in
   std_logic;
      mmio_wr             : in
   std_logic;
      mmio_rd             : in
   std_logic;
      mmio_addr           : in
   std_logic_vector(20 downto 0);
      mmio_wr_data        : in
   std_logic_vector(31 downto 0);
      mmio_rd_data        : out
   std_logic_vector(31 downto 0);
      -- slot interface
      slot_cs_array       : out
   std_logic_vector(63 downto 0);
      slot_mem_rd_array   : out
   std_logic_vector(63 downto 0);
      slot_mem_wr_array   : out
   std_logic_vector(63 downto 0);
      slot_reg_addr_array : out
   slot_2d_reg_type;
      slot_rd_data_array  : in
   slot_2d_data_type;
      slot_wr_data_array  : out
   slot_2d_data_type
   );
end chu_mmio_controller;

architecture arch of
   chu_mmio_controller is
   -- 11 LSBs of address used; 2^6
   slots, each with 2^5 registers
   alias slot_addr :
   std_logic_vector(5 downto 0) is
         mmio_addr(10 downto 5);
   alias reg_addr  :
   std_logic_vector(4 downto 0) is
         mmio_addr(4 downto 0);
begin
   -- address decoding
      process(slot_addr, mmio_cs)
```

```
   begin
      slot_cs_array <= (others =>
   '0');
      if mmio_cs = '1' then

   slot_cs_array(to_integer(unsigned
   (slot_addr))) <= '1';
      end if;
   end process;
   -- broadcast to all slots
   slot_mem_rd_array   <= (others =>
   mmio_rd);
   slot_mem_wr_array   <= (others =>
   mmio_wr);
   slot_wr_data_array  <= (others =>
   mmio_wr_data);
   slot_reg_addr_array <= (others =>
   reg_addr);
   -- mux for read data
   mmio_rd_data <=
   slot_rd_data_array(to_integer(uns
   igned(slot_addr)));
end arch;
```

Listado 3.6. Subsistema MMIO vanilla

```
library ieee;
use ieee.std_logic_1164.all;
use work.chu_io_map.all;
entity mmio_sys_vanilla is
   generic(
     N_LED: integer;
     N_SW: integer
   );
   port(
      -- FPro bus
      clk          : in  std_logic;
      reset        : in  std_logic;
      mmio_cs      : in  std_logic;
      mmio_wr      : in  std_logic;
      mmio_rd      : in  std_logic;
      mmio_addr    : in
   std_logic_vector(20 downto 0); --
   only 11 LSBs used
      mmio_wr_data : in
   std_logic_vector(31 downto 0);
      mmio_rd_data : out
   std_logic_vector(31 downto 0);
      -- switches and LEDs
      sw           : in
   std_logic_vector(N_SW-1 downto
   0);
      led          : out
   std_logic_vector(N_LED-1 downto
   0);
      -- uart
      rx           : in  std_logic;
      tx           : out std_logic
   );
end mmio_sys_vanilla;

architecture arch of
   mmio_sys_vanilla is
   signal cs_array       :
   std_logic_vector(63 downto 0);
   signal reg_addr_array :
   slot_2d_reg_type;
   signal mem_rd_array   :
   std_logic_vector(63 downto 0);
   signal mem_wr_array   :
   std_logic_vector(63 downto 0);
   signal rd_data_array  :
   slot_2d_data_type;
   signal wr_data_array  :
   slot_2d_data_type;
begin
   --
   ********************************
   ********************************
   --  MMIO controller instantiation
   --
   ********************************
   ********************************
   ctrl_unit : entity
   work.chu_mmio_controller
      port map(
         -- FPro bus interface
         mmio_cs             =>
   mmio_cs,
         mmio_wr             =>
   mmio_wr,
         mmio_rd             =>
   mmio_rd,
         mmio_addr           =>
   mmio_addr,
         mmio_wr_data        =>
   mmio_wr_data,
         mmio_rd_data        =>
   mmio_rd_data,
         -- 64 slot interface
         slot_cs_array       =>
   cs_array,
         slot_reg_addr_array =>
   reg_addr_array,
         slot_mem_rd_array   =>
   mem_rd_array,
         slot_mem_wr_array   =>
   mem_wr_array,
         slot_rd_data_array  =>
   rd_data_array,
         slot_wr_data_array  =>
   wr_data_array
      );

   --
   ********************************
   ********************************
   -- IO slots instantiations
   --
   ********************************
   ********************************
   -- slot 0: system timer
   timer_slot0 : entity
   work.chu_timer
      port map(
         clk           => clk,
         reset         => reset,
```

```
      cs            =>
cs_array(S0_SYS_TIMER),
      read          =>
mem_rd_array(S0_SYS_TIMER),
      write         =>
mem_wr_array(S0_SYS_TIMER),
      addr          =>
reg_addr_array(S0_SYS_TIMER),
      rd_data       =>
rd_data_array(S0_SYS_TIMER),
      wr_data       =>
wr_data_array(S0_SYS_TIMER)
   );
-- slot 1: uart1
uart1_slot1 : entity
work.chu_uart
   generic map(FIFO_DEPTH_BIT =>
6)
   port map(
      clk     => clk,
      reset   => reset,
      cs      =>
cs_array(S1_UART1),
      read    =>
mem_rd_array(S1_UART1),
      write   =>
mem_wr_array(S1_UART1),
      addr    =>
reg_addr_array(S1_UART1),
      rd_data =>
rd_data_array(S1_UART1),
      wr_data =>
wr_data_array(S1_UART1),
      -- external signals
      tx      => tx,
      rx      => rx
   );
-- slot 2: GPO for LEDs
gpo_slot2 : entity work.chu_gpo
   generic map(W => N_LED)
   port map(
      clk     => clk,
      reset   => reset,
      cs      =>
cs_array(S2_LED),
      read    =>
mem_rd_array(S2_LED),
      write   =>
mem_wr_array(S2_LED),
      addr    =>
reg_addr_array(S2_LED),
      rd_data =>
rd_data_array(S2_LED),
      wr_data =>
wr_data_array(S2_LED),
      -- external signal
      dout    => led
   );
-- slot 3: input port for
switches
gpi_slot3 : entity work.chu_gpi
   generic map(W => N_SW)
   port map(
      clk     => clk,
      reset   => reset,
      cs      => cs_array(S3_SW),
      read    =>
mem_rd_array(S3_SW),
      write   =>
mem_wr_array(S3_SW),
      addr    =>
reg_addr_array(S3_SW),
      rd_data =>
rd_data_array(S3_SW),
      wr_data =>
wr_data_array(S3_SW),
      -- external signal
      din     => sw
   );
-- assign 0's to all unused slot
rd_data signals
gen_unused_slot : for i in 4 to
63 generate
   rd_data_array(i) <= (others =>
'0');
   end generate gen_unused_slot;
```

Listado 3.7. Puente MCS a FPro

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use work.chu_io_map.all;
entity chu_mcs_bridge is
   generic(BRG_BASE :
   std_logic_vector(31 downto 0) :=
   x"C0000000");
   port(
      -- uBlaze MCS I/O bus
      io_addr_strobe  : in
   std_logic;  -- not used
      io_read_strobe  : in
   std_logic;
      io_write_strobe : in
   std_logic;
      io_byte_enable  : in
   std_logic_vector(3 downto 0);
      io_address      : in
   std_logic_vector(31 downto 0);
      io_write_data   : in
   std_logic_vector(31 downto 0);
      io_read_data    : out
   std_logic_vector(31 downto 0);
      io_ready        : out
   std_logic;
      -- FPro bus
      fp_video_cs     : out
   std_logic;
      fp_mmio_cs      : out
   std_logic;
      fp_wr           : out
   std_logic;
      fp_rd           : out
   std_logic;
      fp_addr         : out
   std_logic_vector(20 downto 0);
      fp_wr_data      : out
   std_logic_vector(31 downto 0);
      fp_rd_data      : in
```

```
   std_logic_vector(31 downto 0)
   );
end chu_mcs_bridge;

architecture arch of chu_mcs_bridge
   is
   signal mcs_bridge_en : std_logic;
   signal word_addr     :
   std_logic_vector(29 downto 0);
begin
   -- address translation and
   decoding
   -- 2 LSBs are "00" due to word
   alignment
   word_addr     <= io_address(31
   downto 2);
   mcs_bridge_en <=
      '1' when io_address(31 downto
   24)=BRG_BASE(31 downto 24) else
   '0';
   fp_video_cs   <=
      '1' when mcs_bridge_en='1' and
   io_address(23)='1' else '0';
   fp_mmio_cs    <=
      '1' when mcs_bridge_en='1' and
   io_address(23)='0' else '0';
   fp_addr       <= word_addr(20
   downto 0);
   -- control line conversion
   fp_wr         <= io_write_strobe;
   fp_rd         <= io_read_strobe;
   io_ready      <= '1'; -- not
   used; transaction done in 1 clock
   -- data line conversion
   fp_wr_data    <= io_write_data;
   io_read_data  <= fp_rd_data;
end arch;
```

Listado 3.8. Sistema FPro vanilla

```
library ieee;
use ieee.std_logic_1164.all;
use work.chu_io_map.all;
entity mcs_top_vanilla is
   generic(BRIDGE_BASE :
   std_logic_vector(31 downto 0) :=
   x"C0000000");
   port(
      clk     : in  std_logic;
      reset_n : in  std_logic;
      -- switches and LEDs
      sw      : in
   std_logic_vector(15 downto 0);
      led     : out
   std_logic_vector(15 downto 0);
      -- uart
      rx      : in  std_logic;
      tx      : out std_logic

   );
end mcs_top_vanilla;

architecture arch of mcs_top_vanilla
   is
   component cpu
      port(
         clk             : in
   std_logic;
         reset           : in
   std_logic;
         io_addr_strobe  : out
   std_logic;
         io_read_strobe  : out
   std_logic;
         io_write_strobe : out
   std_logic;
         io_address      : out
   std_logic_vector(31 downto 0);
         io_byte_enable  : out
   std_logic_vector(3 downto 0);
         io_write_data   : out
   std_logic_vector(31 downto 0);
         io_read_data    : in
   std_logic_vector(31 downto 0);
         io_ready        : in
   std_logic
      );
   end component;
   signal clk_100M        :
   std_logic;
   signal reset_sys       :
   std_logic;
   -- MCS IO bus
   signal io_addr_strobe  :
   std_logic;
   signal io_read_strobe  :
   std_logic;
   signal io_write_strobe :
   std_logic;
   signal io_byte_enable  :
   std_logic_vector(3 downto 0);
   signal io_address      :
   std_logic_vector(31 downto 0);
   signal io_write_data   :
   std_logic_vector(31 downto 0);
   signal io_read_data    :
   std_logic_vector(31 downto 0);
   signal io_ready        :
   std_logic;
   -- fpro bus
   signal fp_mmio_cs         :
   std_logic;
   signal fp_wr          : std_logic;
   signal fp_rd          : std_logic;
   signal fp_addr        :
   std_logic_vector(20 downto 0);
   signal fp_wr_data     :
   std_logic_vector(31 downto 0);
   signal fp_rd_data     :
   std_logic_vector(31 downto 0);
begin
   -- clock and reset
   clk_100M  <= clk;   -- 100 MHz
   external clock
   reset_sys <= not reset_n;
   -- instantiate microBlaze MCS
   mcs_0 : cpu
      port map(
         clk             =>
```

```
clk_100M,
      reset           =>
reset_sys,
      io_addr_strobe  =>
io_addr_strobe,
      io_read_strobe  =>
io_read_strobe,
      io_write_strobe =>
io_write_strobe,
      io_byte_enable  =>
io_byte_enable,
      io_address      =>
io_address,
      io_write_data   =>
io_write_data,
      io_read_data    =>
io_read_data,
      io_ready        => io_ready
   );
-- instantiate MCS IO bus to FPro
bus bridge
bridge_unit : entity
work.chu_mcs_bridge
   generic map(BRG_BASE =>
BRIDGE_BASE)
   port map(
      io_addr_strobe  =>
io_addr_strobe,
      io_read_strobe  =>
io_read_strobe,
      io_write_strobe =>
io_write_strobe,
      io_byte_enable  =>
io_byte_enable,
      io_address      =>
io_address,
      io_write_data   =>
io_write_data,
      io_read_data    =>
io_read_data,
      io_ready        =>
io_ready,
      fp_video_cs     => open,
      fp_mmio_cs      =>
fp_mmio_cs,
      fp_wr           => fp_wr,
      fp_rd           => fp_rd,
      fp_addr         => fp_addr,
      fp_wr_data      =>
fp_wr_data,
      fp_rd_data      =>
fp_rd_data
   );
-- instantiate vanilla MMIO
subsystem
mmio_sys_unit : entity
work.mmio_sys_vanilla
   generic map(
      N_LED=>16,
      N_SW=>16
   )
   port map(
      clk          => clk_100M,
      reset        => reset_sys,
      mmio_cs      => fp_mmio_cs,
      mmio_wr      => fp_wr,
      mmio_rd      => fp_rd,
      mmio_addr    => fp_addr,
      mmio_wr_data => fp_wr_data,
      mmio_rd_data => fp_rd_data,
      sw           => sw,
      led          => led,
      rx           => rx,
      tx           => tx
   );
end arch;
```

Listado 4.1. Generador de velocidad

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity baud_gen is
   port(
      clk   : in std_logic;
      reset : in std_logic;
      dvsr  : in std_logic_vector(10
   downto 0);
      tick  : out std_logic
   );
end baud_gen;

architecture arch of baud_gen is
   constant N    : integer := 11;
   signal r_reg  : unsigned(N - 1
   downto 0);
   signal r_next : unsigned(N - 1
   downto 0);
begin
   -- register
   process(clk, reset)
   begin
      if (reset = '1') then
         r_reg <= (others => '0');
      elsif (clk'event and clk =
   '1') then
         r_reg <= r_next;
      end if;
   end process;
   -- next-state logic
   r_next <= (others=>'0') when
   r_reg=unsigned(dvsr) else r_reg +
   1;
   -- output logic
   tick <= '1' when r_reg=1 else
   '0'; -- not use 0 because of
   reset
end arch;
```

Listado 4.2. Controlador MMIO

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use work.chu_io_map.all;
```

```
entity chu_mmio_controller is
   port(
      -- FPro bus
      mmio_cs              : in
   std_logic;
      mmio_wr              : in
   std_logic;
      mmio_rd              : in
   std_logic;
      mmio_addr            : in
   std_logic_vector(20 downto 0);
      mmio_wr_data         : in
   std_logic_vector(31 downto 0);
      mmio_rd_data         : out
   std_logic_vector(31 downto 0);
      -- slot interface
      slot_cs_array        : out
   std_logic_vector(63 downto 0);
      slot_mem_rd_array    : out
   std_logic_vector(63 downto 0);
      slot_mem_wr_array    : out
   std_logic_vector(63 downto 0);
      slot_reg_addr_array : out
   slot_2d_reg_type;
      slot_rd_data_array  : in
   slot_2d_data_type;
      slot_wr_data_array  : out
   slot_2d_data_type
   );
end chu_mmio_controller;

architecture arch of
   chu_mmio_controller is
   -- 11 LSBs of address used; 2^6
   slots, each with 2^5 registers
   alias slot_addr :
   std_logic_vector(5 downto 0) is
         mmio_addr(10 downto 5);
   alias reg_addr  :
   std_logic_vector(4 downto 0) is
         mmio_addr(4 downto 0);
begin
   -- address decoding
      process(slot_addr, mmio_cs)
   begin
      slot_cs_array <= (others =>
   '0');
      if mmio_cs = '1' then

   slot_cs_array(to_integer(unsigned
   (slot_addr))) <= '1';
      end if;
   end process;
   -- broadcast to all slots
   slot_mem_rd_array   <= (others =>
   mmio_rd);
   slot_mem_wr_array   <= (others =>
   mmio_wr);
   slot_wr_data_array  <= (others =>
   mmio_wr_data);
   slot_reg_addr_array <= (others =>
   reg_addr);
   -- mux for read data
   mmio_rd_data <=
   slot_rd_data_array(to_integer(uns
      igned(slot_addr)));
end arch;
```

Listado 4.3. Transmisor UART

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity uart_tx is
   generic(
      DBIT    : integer := 8;    -- #
   data bits
      SB_TICK : integer := 16    -- #
   ticks for stop bits
   );
   port(
      clk, reset   : in  std_logic;
      tx_start     : in  std_logic;
      s_tick       : in  std_logic;
      din          : in
   std_logic_vector(7 downto 0);
      tx_done_tick : out std_logic;
      tx           : out std_logic
   );
end uart_tx;

architecture arch of uart_tx is
   type state_type is (idle, start,
   data, stop);
   signal state_reg       :
   state_type;
   signal state_next      :
   state_type;
   signal s_reg, s_next   :
   unsigned(4 downto 0);
   signal n_reg, n_next   :
   unsigned(2 downto 0);
   signal b_reg, b_next   :
   std_logic_vector(7 downto 0);
   signal tx_reg, tx_next :
   std_logic;
begin
   -- FSMD state & data registers
   process(clk, reset)
   begin
      if reset = '1' then
         state_reg <= idle;
         s_reg     <= (others =>
   '0');
         n_reg     <= (others =>
   '0');
         b_reg     <= (others =>
   '0');
         tx_reg    <= '1';
      elsif (clk'event and clk =
   '1') then
         state_reg <= state_next;
         s_reg     <= s_next;
         n_reg     <= n_next;
         b_reg     <= b_next;
         tx_reg    <= tx_next;
      end if;
   end process;
   -- next-state logic & data path
```

```
process(state_reg,s_reg,n_reg,b_r
eg,s_tick,tx_reg,tx_start,din)
begin
   state_next   <= state_reg;
   s_next       <= s_reg;
   n_next       <= n_reg;
   b_next       <= b_reg;
   tx_next      <= tx_reg;
   tx_done_tick <= '0';
   case state_reg is
      when idle =>
         tx_next <= '1';
         if tx_start = '1' then
            state_next <= start;
            s_next     <= (others
=> '0');
            b_next     <= din;
         end if;
      when start =>
         tx_next <= '0';
         if (s_tick = '1') then
            if s_reg = 15 then
               state_next <=
data;
               s_next     <=
(others => '0');
               n_next     <=
(others => '0');
            else
               s_next <= s_reg +
1;
            end if;
         end if;
      when data =>
         tx_next <= b_reg(0);
         if (s_tick = '1') then
            if s_reg = 15 then
               s_next <= (others
=> '0');
               b_next <= '0' &
b_reg(7 downto 1);
               if n_reg = (DBIT -
1) then
                  state_next <=
stop;
               else
                  n_next <= n_reg
+ 1;
               end if;
            else
               s_next <= s_reg +
1;
            end if;
         end if;
      when stop =>
         tx_next <= '1';
         if (s_tick = '1') then
            if s_reg = (SB_TICK -
1) then
               state_next   <=
idle;
               tx_done_tick <=
'1';
            else
               s_next <= s_reg +
1;
            end if;
         end if;
      end case;
   end process;
   tx <= tx_reg;
end arch;
```

Listado 4.4. Descripción UART de alto nivel

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity uart is
   generic(
      DBIT    : integer := 8;   -- #
   data bits
      SB_TICK : integer := 16;  -- #
   ticks for stop bits, 16 per bit
      FIFO_W  : integer := 4    -- #
   FIFO addr bits (depth: 2^FIFO_W)
   );
   port(
      clk, reset : in  std_logic;
      rd_uart    : in  std_logic;
      wr_uart    : in  std_logic;
      dvsr       : in
   std_logic_vector(10 downto 0);
      rx         : in  std_logic;
      w_data     : in
   std_logic_vector(7 downto 0);
      tx_full    : out std_logic;
      rx_empty   : out std_logic;
      r_data     : out
   std_logic_vector(7 downto 0);
      tx         : out std_logic
   );
end uart;

architecture str_arch of uart is
   signal tick             :
   std_logic;
   signal rx_done_tick     :
   std_logic;
   signal tx_fifo_out      :
   std_logic_vector(7 downto 0);
   signal rx_data_out      :
   std_logic_vector(7 downto 0);
   signal tx_empty         :
   std_logic;
   signal tx_fifo_not_empty :
   std_logic;
   signal tx_done_tick     :
   std_logic;
begin
   baud_gen_unit : entity
   work.baud_gen(arch)
      port map(
         clk  => clk, reset =>
   reset, dvsr => dvsr,
         tick => tick
```

```
    );
  uart_rx_unit : entity
  work.uart_rx(arch)
    generic map(DBIT => DBIT,
  SB_TICK => SB_TICK)
    port map(
      clk          => clk,
      reset        => reset,
      rx           => rx,
      s_tick       => tick,
      rx_done_tick =>
  rx_done_tick,
      dout         => rx_data_out
    );
  uart_tx_unit : entity
  work.uart_tx(arch)
    generic map(DBIT => DBIT,
  SB_TICK => SB_TICK)
    port map(
      clk          => clk,
      reset        => reset,
      tx_start     =>
  tx_fifo_not_empty,
      s_tick       => tick,
      din          =>
  tx_fifo_out,
      tx_done_tick =>
  tx_done_tick,
      tx           => tx
    );
  fifo_rx_unit : entity
  work.fifo(reg_file_arch)
    generic map(DATA_WIDTH =>
  DBIT, ADDR_WIDTH => FIFO_W)
    port map(
      clk    => clk,
      reset  => reset,
      rd     => rd_uart,
      wr     => rx_done_tick,
      w_data => rx_data_out,
      empty  => rx_empty,
      full   => open,
      r_data => r_data
    );
  fifo_tx_unit : entity
  work.fifo(reg_file_arch)
    generic map(DATA_WIDTH =>
  DBIT, ADDR_WIDTH => FIFO_W)
    port map(
      clk    => clk,
      reset  => reset,
      rd     => tx_done_tick,
      wr     => wr_uart,
      w_data => w_data,
      empty  => tx_empty,
      full   => tx_full,
      r_data => tx_fifo_out
    );

  tx_fifo_not_empty <= not
  tx_empty;
end str_arch;
```

Listado 4.5. Núcleo UART

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity chu_uart is
  generic(
    FIFO_DEPTH_BIT : integer := 8
  -- # FIFO addr bits
  );
  port(
    clk     : in  std_logic;
    reset   : in  std_logic;
    -- slot interface
    cs      : in  std_logic;
    write   : in  std_logic;
    read    : in  std_logic;
    addr    : in
  std_logic_vector(4 downto 0);
    rd_data : out
  std_logic_vector(31 downto 0);
    wr_data : in
  std_logic_vector(31 downto 0);
    -- external signals
    tx      : out std_logic;
    rx      : in  std_logic
  );
end chu_uart;

architecture arch of chu_uart is
  signal wr_en    : std_logic;
  signal wr_uart  : std_logic;
  signal rd_uart  : std_logic;
  signal wr_dvsr  : std_logic;
  signal tx_full  : std_logic;
  signal rx_empty : std_logic;
  signal r_data   :
  std_logic_vector(7 downto 0);
  signal dvsr_reg :
  std_logic_vector(10 downto 0);
begin
  -- instantiate uart controller
  uart_unit : entity
  work.uart(str_arch)
    generic map(
      DBIT    => 8,
      SB_TICK => 16,
      FIFO_W  => FIFO_DEPTH_BIT
    )
    port map(
      clk      => clk,
      reset    => reset,
      rd_uart  => rd_uart,
      wr_uart  => wr_uart,
      dvsr     => dvsr_reg,
      rx       => rx,
      tx       => tx,
      w_data   => wr_data(7
  downto 0),
      r_data   => r_data,
      tx_full  => tx_full,
      rx_empty => rx_empty
    );
  -- baud rate register
  process(clk, reset)
```

```
   begin
      if (reset = '1') then
         dvsr_reg <= (others =>
   '0');
      elsif (clk'event and clk =
   '1') then
         if wr_dvsr = '1' then
            dvsr_reg <= wr_data(10
   downto 0);
         end if;
      end if;
   end process;
   -- write decoding
   wr_en   <= '1' when write = '1'
   and cs = '1' else '0';
   wr_dvsr <= '1' when addr(1 downto
   0)="01" and wr_en = '1' else '0';
   wr_uart <= '1' when addr(1 downto
   0)="10" and wr_en = '1' else '0';
   rd_uart <= '1' when addr(1 downto
   0)="11" and wr_en = '1' else '0';
   -- read multiplexing
   rd_data <= x"00000" & "00" &
   tx_full & rx_empty & r_data;
end arch;
```

Listado 4.6. Definición de clase UartCore (uart_core.h)

```
#ifndef _UART_CORE_H_INCLUDED
#define _UART_CORE_H_INCLUDED

#include "chu_io_rw.h"
#include "chu_io_map.h"  // to use
   SYS_CLK_FREQ
class UartCore {
   /* Register map */
   enum {
      RD_DATA_REG = 0,   /**< rx
   data/status register */
      DVSR_REG = 1,      /**< baud
   rate diviser register */
      WR_DATA_REG = 2,   /**< wr
   data register */
      RM_RD_DATA_REG = 3 /**< remove
   read data offset */
   };
  /* mask fields */
   enum {
      TX_FULL_FIELD = 0x00000200,
   /**< bit 9 of rd_data_reg; full
   bit  */
      RX_EMPT_FIELD = 0x00000100,
   /**< bit 10 of rd_data_reg; empty
   bit */
      RX_DATA_FIELD = 0x000000ff
   /**< bits 7..0 rd_data_reg; read
   data */
   };
public:
   /* methods */
   UartCore(uint32_t
   core_base_addr);
   ~UartCore();
   //basic I/O access
   void set_baud_rate(int baud);
   int rx_fifo_empty();
   int tx_fifo_full();
   void tx_byte(uint8_t byte);
   int rx_byte();
   //display methods
   void disp(char ch);
   void disp(double f, int digit);
   void disp(double f);
private:
   uint32_t base_addr;
   int baud_rate;
   void disp_str(const char *str);
};
#endif  // _UART_CORE_H_INCLUDED
```

Listado 4.7. Métodos básicos UartCore (uart_core.cpp)

```
UartCore::UartCore(uint32_t
   core_base_addr) {
   base_addr = core_base_addr;
   set_baud_rate(9600);
   //default baud rate
}

UartCore::~UartCore() {
}

/* baud rate = sys_freq/16/(dvsr+1)
   */
void UartCore::set_baud_rate(int
   baud) {
   uint32_t dvsr;

   dvsr = SYS_CLK_FREQ*1000000 / 16
   / baud - 1;
   io_write(base_addr, DVSR_REG,
   dvsr);
}

int UartCore::rx_fifo_empty() {
   uint32_t rd_word;
   int empty;

   rd_word = io_read(base_addr,
   RD_DATA_REG);
   empty = (int) (rd_word &
   RX_EMPT_FIELD) >> 8;
   return (empty);
}

int UartCore::tx_fifo_full() {
   uint32_t rd_word;
   int full;

   rd_word = io_read(base_addr,
   RD_DATA_REG);
   full = (int) (rd_word &
   TX_FULL_FIELD) >> 9;
   return (full);
}
```

```
void UartCore::tx_byte(uint8_t byte)
   {
   while (tx_fifo_full()) {
   };  // busy waiting
   io_write(base_addr, WR_DATA_REG,
   (uint32_t )byte);
}

int UartCore::rx_byte() {
   uint32_t data;

   if (rx_fifo_empty())
      return (-1);
   else {
      data = io_read(base_addr,
   RD_DATA_REG) & RX_DATA_FIELD;
      io_write(base_addr,
   RM_RD_DATA_REG, 0); //dummy write
   to remove data from rx FIFO
      return ((int) data);
   }
}
```

Listado 4.8. Métodos de visualización UartCore (uart_core.cpp)

```
void UartCore::disp_str(const char
   *str) {
   while ((uint8_t) *str) {
      tx_byte(*str);
      str++;
   }
}
void UartCore::disp(const char *str)
   {
   disp_str(str);
}

void UartCore::disp(char ch) {
    tx_byte(ch);
}

void UartCore::disp(int n, int base,
   int len) {
   char buf[33];         // 32 bit #
   char *str, ch, sign;
   int rem, i;
   unsigned int un;
   /* error check */
   if (base != 2 && base != 8 &&
   base != 16)
      base = 10;
   if (len > 32)  // error check
      len = 32;
   /* handle neg decimal # */
   if (base == 10 && n < 0) {
      un = (unsigned) -n;
      sign = '-';
   } else {
      un = (unsigned) n; //
   interpreted as unsigned for
   hex/bin conversion
      sign = ' ';
   }
   /* convert # to string */
   str = &buf[33];
   *str = '\0';
   i = 0;
   do {
      str--;
      rem = un % base;
      un = un / base;
      if (rem < 10)
         ch = (char) rem + '0';
      else
         ch = (char) rem - 10 + 'a';
      *str = ch;
      i++;
   } while (un);
   /* attach - sign for neg decimal
   # */
   if (sign == '-') {
      str--;
      *str = sign;
      i++;
   }
   /* pad with blank */
   while (i < len) {
      str--;
      *str = ' ';
      i++;
   };
   disp_str(str);
}
void UartCore::disp(int n) {
   disp(n, 10, 0);
}
void UartCore::disp(int n, int base)
   {
   disp(n, base, 0);
}
void UartCore::disp(double f, int
   digit) {
   double fa, frac; // absolute
   value of f
   int n, i, i_part;

   fa = f;
   if (f < 0.0) {
      fa = -f;
      disp_str("-");
   }
   // display integer portion
   i_part = (int) fa; // integer
   part of f
   disp(i_part);
   disp_str(".");
   // display fraction part
   frac = fa - (double) i_part;
   for (n = 0; n < digit; n++) {
    frac = frac * 10.0;
      i = (int) frac;
      disp(i);
      frac = frac - i;
   }
}
```

```
void UartCore::disp(double f) {
   disp(f, 3);
}
```

Listado 5.1. Núcleo XADC

```
library ieee;
use ieee.std_logic_1164.all;
entity chu_xadc_core is
   port(
      clk     : in  std_logic;
      reset   : in  std_logic;
      -- slot interface
      cs      : in  std_logic;
      write   : in  std_logic;
      read    : in  std_logic;
      addr    : in
   std_logic_vector(4 downto 0);
      rd_data : out
   std_logic_vector(31 downto 0);
      wr_data : in
   std_logic_vector(31 downto 0);
      -- external signals
      adc_p   : in
   std_logic_vector(3 downto 0);
      adc_n   : in
   std_logic_vector(3 downto 0)
   );
end chu_xadc_core;

architecture arch of chu_xadc_core
   is
   component xadc_fpro
      port(
         di_in       : in
   std_logic_vector(15 downto 0);
         daddr_in    : in
   std_logic_vector(6 downto 0);
         den_in      : in
   std_logic;
         dwe_in      : in
   std_logic;
         drdy_out    : out
   std_logic;
         do_out      : out
   std_logic_vector(15 downto 0);
         dclk_in     : in
   std_logic;
         reset_in    : in
   std_logic;
         vp_in       : in
   std_logic;
         vn_in       : in
   std_logic;
         vauxp2      : in
   std_logic;
         vauxn2      : in
   std_logic;
         vauxp3      : in
   std_logic;
         vauxn3      : in
   std_logic;
         vauxp10     : in
   std_logic;
         vauxn10     : in
   std_logic;
         vauxp11     : in
   std_logic;
         vauxn11     : in
   std_logic;
         channel_out : out
   std_logic_vector(4 downto 0);
         eoc_out     : out
   std_logic;
         alarm_out   : out
   std_logic;
         eos_out     : out
   std_logic;
         busy_out    : out std_logic
      );
   end component;
   signal channel      :
   std_logic_vector(4 downto 0);
   signal daddr_in     :
   std_logic_vector(6 downto 0);
   signal eoc          : std_logic;
   signal rdy          : std_logic;
   signal adc_data     :
   std_logic_vector(15 downto 0);
   signal adc0_out_reg :
   std_logic_vector(15 downto 0);
   signal adc1_out_reg :
   std_logic_vector(15 downto 0);
   signal adc2_out_reg :
   std_logic_vector(15 downto 0);
   signal adc3_out_reg :
   std_logic_vector(15 downto 0);
   signal tmp_out_reg  :
   std_logic_vector(15 downto 0);
   signal vcc_out_reg  :
   std_logic_vector(15 downto 0);
begin
   -- instantiate cusomized xadc
   core
   xdac_unit : xadc_fpro
      port map(
         dclk_in     => clk,
         reset_in    => reset,
   --reset,
         di_in       => (others =>
   '0'),
         daddr_in    => daddr_in,
         den_in      => eoc,
         dwe_in      => '0',
   -- read only
         drdy_out    => rdy,
         do_out      => adc_data,
         vp_in       => '0',
         vn_in       => '0',
         vauxp2      => adc_p(2),
         vauxn2      => adc_n(2),
         vauxp3      => adc_p(0),
         vauxn3      => adc_n(0),
         vauxp10     => adc_p(1),
         vauxn10     => adc_n(1),
         vauxp11     => adc_p(3),
         vauxn11     => adc_n(3),
         channel_out => channel,
```

```
      eoc_out     => eoc,
      eos_out     => open,
      busy_out    => open,
      alarm_out   => open
   );
-- form xadc DRP address
daddr_in <= "00" & channel;
-- registers and decoding
process(clk, reset)
begin
   if reset = '1' then
      adc0_out_reg <= (others =>
'0');
      adc1_out_reg <= (others =>
'0');
      adc2_out_reg <= (others =>
'0');
      adc3_out_reg <= (others =>
'0');
      tmp_out_reg  <= (others =>
'0');
      vcc_out_reg  <= (others =>
'0');
   elsif (clk'event and clk =
'1') then
      if rdy = '1' and channel =
"10011" then
         adc0_out_reg <=
adc_data;
      end if;
      if rdy = '1' and channel =
"11010" then
         adc1_out_reg <=
adc_data;
      end if;
      if rdy = '1' and channel =
"10010" then
         adc2_out_reg <=
adc_data;
      end if;
      if rdy = '1' and channel =
"11011" then
         adc3_out_reg <=
adc_data;
      end if;
      if rdy = '1' and channel =
"00000" then
         tmp_out_reg <= adc_data;
      end if;
      if rdy = '1' and channel =
"00001" then
         vcc_out_reg <= adc_data;
      end if;
   end if;
end process;
-- read multiplexing
with addr(2 downto 0) select
   rd_data <=
      x"0000" & adc0_out_reg when
"000",
      x"0000" & adc1_out_reg when
"001",
      x"0000" & adc2_out_reg when
"010",
      x"0000" & adc3_out_reg when
   "011",
         x"0000" & tmp_out_reg  when
   "100",
         x"0000" & vcc_out_reg  when
   others;
end arch;
```

Listado 5.2. Definición de clase

```
   XadcCore (xadc_core.h)
#ifndef _XADC_CORE_H_INCLUDED
#define _XADC_CORE_H_INCLUDED

#include "chu_init.h"
class XadcCore {
public:
   /* Register map */
   enum {
      ADC_0_REG = 0, /**< 16-bit
   data from Nexys-4 adc input #0
   */
      TMP_REG   = 4,   /**< FPGA
   internal temperature */
      VCC_REG   = 5,   /**< FPGA
   internal core volatge */
   };

   /* Constructor */
   XadcCore(uint32_t
   core_base_addr);
   ~XadcCore(); // not used
   uint16_t read_raw(int n);
   double read_adc_in(int n);
   double read_fpga_vcc();
   double read_fpga_temp();
private:
   uint32_t base_addr;
};
```

Listado 5.3. Implementación de clase XadcCore (xadc_core.cpp)

```
XadcCore::XadcCore(uint32_t
   core_base_addr) {
   base_addr = core_base_addr;
}

XadcCore::~XadcCore() {
}

uint16_t XadcCore::read_raw(int n) {
   uint16_t rd_data;

   rd_data = (uint16_t)
   io_read(base_addr, n) &
   0x0000ffff;
   return (rd_data);
}

double XadcCore::read_adc_in(int n)
   {
   uint16_t raw;
   raw = read_raw(n) >> 4;
```

```
   return ((double) raw / 4096.0);
}

// input source 5 is connected to
   vcc reading
double XadcCore::read_fpga_vcc() {
   return (read_adc_in(VCC_REG) *
   3.0);
}

// input source 4 is connected to
   temperature reading
double XadcCore::read_fpga_temp() {
   return (read_adc_in(TMP_REG) *
   503.975 - 273.15);
```

Listado 5.4. Función de prueba XADC (main_sampler_test.cpp)

```
void adc_check(XadcCore *adc_p,
   GpoCore *led_p) {
   double reading;
   int n, i;
   uint16_t raw;

   for (i = 0; i < 5; i++) {
      // display 12-bit channel 0
   reading in LED
      raw = adc_p->read_raw(0);
      raw = raw >> 4;
      led_p->write(raw);
      // display on-chip sensor and
   4 channels in console
      uart.disp("FPGA vcc/temp: ");
      reading = adc_p-
   >read_fpga_vcc();
      uart.disp(reading, 3);
      uart.disp(" / ");
      reading = adc_p-
   >read_fpga_temp();
      uart.disp(reading, 3);
      uart.disp("\n\r");
      for (n = 0; n < 4; n++) {
         uart.disp("analog
   channel/voltage: ");
         uart.disp(n);
         uart.disp(" / ");
         reading = adc_p-
   >read_adc_in(n);
         uart.disp(reading, 3);
         uart.disp("\n\r");
      } // end for
      sleep_ms(200);
   }
}
```

Listado 5.5. Subsistema de prueba MMIO

```
library ieee;
use ieee.std_logic_1164.all;
use work.chu_io_map.all;
entity mmio_sys_sampler is
   port(
      -- FPro bus
      clk          : in
   std_logic;
      reset        : in
   std_logic;
      mmio_cs      : in
   std_logic;
      mmio_wr      : in
   std_logic;
      mmio_rd      : in
   std_logic;
      mmio_addr    : in
   std_logic_vector(20 downto 0);
      mmio_wr_data : in
   std_logic_vector(31 downto 0);
      mmio_rd_data : out
   std_logic_vector(31 downto 0);
      -- switches and LEDs
      sw           : in
   std_logic_vector(15 downto 0);
      led          : out
   std_logic_vector(15 downto 0);
      -- uart
      rx           : in
   std_logic;
      tx           : out
   std_logic;
        -- 4 analog input pair
      adc_p        : in
   std_logic_vector(3 downto 0);
      adc_n        : in
   std_logic_vector(3 downto 0);
      -- pwm
      pwm          : out
   std_logic_vector(7 downto 0);
      -- btn
      btn          : in
   std_logic_vector(4 downto 0);
      -- 8-digit 7-seg LEDs
      an           : out
   std_logic_vector(7 downto 0);
      sseg         : out
   std_logic_vector(7 downto 0)
      -- spi accelerator

      acl_sclk     : out
   std_logic;
      acl_mosi     : out
   std_logic;
      acl_miso     : in
   std_logic;
      acl_ss       : out
   std_logic;
      -- i2c temperature sensor
      tmp_i2c_scl  : out
   std_logic;
      tmp_i2c_sda  : inout
   std_logic;
      -- ps2
      ps2d         : inout
   std_logic;
      ps2c         : inout
```

```
   std_logic;
      --ddfs square wave output
      ddfs_sq_wave : out
   std_logic;
      -- 1-bit dac
      pdm          : out   std_logic
   );
end mmio_sys_sampler;

architecture arch of
   mmio_sys_sampler is
   signal cs_array       :
   std_logic_vector(63 downto 0);
   signal reg_addr_array :
   slot_2d_reg_type;
   signal mem_rd_array   :
   std_logic_vector(63 downto 0);
   signal mem_wr_array   :
   std_logic_vector(63 downto 0);
   signal rd_data_array  :
   slot_2d_data_type;
   signal wr_data_array  :
   slot_2d_data_type;
   signal adsr_env       :
   std_logic_vector(15 downto 0);
begin
   --
   ********************************
   ********************************
   --  MMIO controller instantiation
   --
   ********************************
   ********************************
   ctrl_unit : entity
   work.chu_mmio_controller
      port map(
         -- FPro bus interface
         mmio_cs             =>
   mmio_cs,
         mmio_wr             =>
   mmio_wr,
         mmio_rd             =>
   mmio_rd,
         mmio_addr           =>
   mmio_addr,
         mmio_wr_data        =>
   mmio_wr_data,
         mmio_rd_data        =>
   mmio_rd_data,
         -- 64 slot interface
         slot_cs_array       =>
   cs_array,
         slot_reg_addr_array =>
   reg_addr_array,
         slot_mem_rd_array   =>
   mem_rd_array,
         slot_mem_wr_array   =>
   mem_wr_array,
         slot_rd_data_array  =>
   rd_data_array,
         slot_wr_data_array  =>
   wr_data_array
      );
   --
   ********************************
   ********************************
   -- IO slots instantiations
   --
   ********************************
   ********************************
   -- slot 0: system timer
   timer_slot0 : entity
   work.chu_timer
      port map(
         clk     => clk,
         reset   => reset,
         cs      =>
   cs_array(S0_SYS_TIMER),
         read    =>
   mem_rd_array(S0_SYS_TIMER),
         write   =>
   mem_wr_array(S0_SYS_TIMER),
         addr    =>
   reg_addr_array(S0_SYS_TIMER),
         rd_data =>
   rd_data_array(S0_SYS_TIMER),
         wr_data =>
   wr_data_array(S0_SYS_TIMER)
      );
   -- slot 1: uart1
   uart1_slot1 : entity
   work.chu_uart
      generic map(FIFO_DEPTH_BIT =>
   6)
      port map(
         clk     => clk,
         reset   => reset,
         cs      =>
   cs_array(S1_UART1),
         read    =>
   mem_rd_array(S1_UART1),
         write   =>
   mem_wr_array(S1_UART1),
         addr    =>
   reg_addr_array(S1_UART1),
         rd_data =>
   rd_data_array(S1_UART1),
         wr_data =>
   wr_data_array(S1_UART1),
         -- external signals
         tx      => tx,
         rx      => rx
      );
   -- slot 2: GPO for 16 LEDs
   gpo_slot2 : entity work.chu_gpo
      generic map(W => 16)
      port map(
         clk     => clk,
         reset   => reset,
         cs      =>
   cs_array(S2_LED),
         read    =>
   mem_rd_array(S2_LED),
         write   =>
   mem_wr_array(S2_LED),
         addr    =>
   reg_addr_array(S2_LED),
         rd_data =>
   rd_data_array(S2_LED),
         wr_data =>
```

```
   wr_data_array(S2_LED),
         -- external signal
         dout    => led
      );
   -- slot 3: input port for 16
   slide switches
   gpi_slot3 : entity work.chu_gpi
      generic map(W => 16)
      port map(
         clk     => clk,
         reset   => reset,
         cs      => cs_array(S3_SW),
         read    =>
   mem_rd_array(S3_SW),
         write   =>
   mem_wr_array(S3_SW),
         addr    =>
   reg_addr_array(S3_SW),
         rd_data =>
   rd_data_array(S3_SW),
         wr_data =>
   wr_data_array(S3_SW),
         -- external signal
         din     => sw
      );
   -- slot 4: reserved for user
   defined
   -- user_slot4 : entity work.
   -- port map(
   --    clk      => clk,
   --    reset    => reset,
   --    cs       =>
   cs_array(S4_USER),
   --    read     =>
   mem_rd_array(S4_USER),
   --    write    =>
   mem_wr_array(S4_USER),
   --    addr     =>
   reg_addr_array(S4_USER),
   --    rd_data  =>
   rd_data_array(S4_USER),
   --    wr_data  =>
   wr_data_array(S4_USER)
   -- );
--   rd_data_array(4) <= (others =>
   '0');
   -- slot 5: xadc
   xadc_slot5 : entity
   work.chu_xadc_core
      port map(
         clk     => clk,
         reset   => reset,
         cs      =>
   cs_array(S5_XADC),
         read    =>
   mem_rd_array(S5_XADC),
         write   =>
   mem_wr_array(S5_XADC),
         addr    =>
   reg_addr_array(S5_XADC),
         rd_data =>
   rd_data_array(S5_XADC),
         wr_data =>
   wr_data_array(S5_XADC),
         -- external signal
         adc_p   => adc_p,
         adc_n   => adc_n
      );
--   -- slot 6: pwm
   pwm_slot6 : entity
   work.chu_io_pwm_core
      generic map(
         W => 8,
         R => 10)
      port map(
         clk     => clk,
         reset   => reset,
         cs      =>
   cs_array(S6_PWM),
         read    =>
   mem_rd_array(S6_PWM),
         write   =>
   mem_wr_array(S6_PWM),
         addr    =>
   reg_addr_array(S6_PWM),
         rd_data =>
   rd_data_array(S6_PWM),
         wr_data =>
   wr_data_array(S6_PWM),
         -- external interface
         pwm_out => pwm
      );
   -- slot 7: push button
   debounce_slot7 : entity
   work.chu_debounce_core
      generic map(
         W => 5,
         N => 20
      )
      port map(
         clk     => clk,
         reset   => reset,
         cs      =>
   cs_array(S7_BTN),
         read    =>
   mem_rd_array(S7_BTN),
         write   =>
   mem_wr_array(S7_BTN),
         addr    =>
   reg_addr_array(S7_BTN),
         rd_data =>
   rd_data_array(S7_BTN),
         wr_data =>
   wr_data_array(S7_BTN),
         -- external interface
         din     => btn
      );
--   -- slot 8: 7-seg LED
   sseg_led_slot8 : entity
   work.chu_led_mux_core
      port map(
         clk     => clk,
         reset   => reset,
         cs      =>
   cs_array(S8_SSEG),
         read    =>
   mem_rd_array(S8_SSEG),
         write   =>
   mem_wr_array(S8_SSEG),
         addr    =>
```

```
   reg_addr_array(S8_SSEG),
         rd_data =>
   rd_data_array(S8_SSEG),
         wr_data =>
   wr_data_array(S8_SSEG),
         -- external interface
         an      => an,
         sseg    => sseg
      );
--   -- slot 9 SPI
--   spi_slot9 : entity
   work.chu_spi_core
--      generic map(S => 1)
--      port map(
--         clk         => clk,
--         reset       => reset,
--         cs          =>
   cs_array(S9_SPI),
--         read        =>
   mem_rd_array(S9_SPI),
--         write       =>
   mem_wr_array(S9_SPI),
--         addr        =>
   reg_addr_array(S9_SPI),
--         rd_data     =>
   rd_data_array(S9_SPI),
--         wr_data     =>
   wr_data_array(S9_SPI),
--         spi_sclk    => acl_sclk,
--         spi_mosi    => acl_mosi,
--         spi_miso    => acl_miso,
--         spi_ss_n(0) => acl_ss
--      );
--   -- slot 10: i2C
--   i2c_slot10 : entity
   work.chu_i2c_core
--      port map(
--         clk     => clk,
--         reset   => reset,
--         cs      =>
   cs_array(S10_I2C),
--         read    =>
   mem_rd_array(S10_I2C),
--         write   =>
   mem_wr_array(S10_I2C),
--         addr    =>
   reg_addr_array(S10_I2C),
--         rd_data =>
   rd_data_array(S10_I2C),
--         wr_data =>
   wr_data_array(S10_I2C),
--         scl     => tmp_i2c_scl,
--         sda     => tmp_i2c_sda
--      );
--   -- slot 11: PS2
--   ps2_slot11 : entity
   work.chu_ps2_core
--      generic map(W_SIZE => 6)
--      port map(
--         clk     => clk,
--         reset   => reset,
--         cs      =>
   cs_array(S11_PS2),
--         read    =>
   mem_rd_array(S11_PS2),
--         write   =>
   mem_wr_array(S11_PS2),
--         addr    =>
   reg_addr_array(S11_PS2),
--         rd_data =>
   rd_data_array(S11_PS2),
--         wr_data =>
   wr_data_array(S11_PS2),
--         -- external interface
--         ps2d    => ps2d,
--         ps2c    => ps2c
--      );
--   -- slot 12: ddfs
--   ddfs_slot12 : entity
   work.chu_ddfs_core
--      generic map(PW => 30)
--      port map(
--         clk         => clk,
--         reset       => reset,
--         cs          =>
   cs_array(S12_DDFS),
--         read        =>
   mem_rd_array(S12_DDFS),
--         write       =>
   mem_wr_array(S12_DDFS),
--         addr        =>
   reg_addr_array(S12_DDFS),
--         rd_data     =>
   rd_data_array(S12_DDFS),
--         wr_data     =>
   wr_data_array(S12_DDFS),
--         -- external interface
--         focw_ext    => (others =>
   '0'),
--         pha_ext     => (others =>
   '0'),
--         env_ext     => adsr_env,
--         pcm_out     => open,
--         digital_out =>
   ddfs_sq_wave,
--         pdm_out     => pdm
--      );
--   -- slot 13: adsr
--   adsr_slot13 : entity
   work.chu_adsr_core
--      port map(
--         clk      => clk,
--         reset    => reset,
--         cs       =>
   cs_array(S13_ADSR),
--         read     =>
   mem_rd_array(S13_ADSR),
--         write    =>
   mem_wr_array(S13_ADSR),
--         addr     =>
   reg_addr_array(S13_ADSR),
--         rd_data  =>
   rd_data_array(S13_ADSR),
--         wr_data  =>
   wr_data_array(S13_ADSR),
--         -- external interface
--         adsr_env => adsr_env
--      );
   -- assign 0's to all unused slot
   rd_data signals
```

```
   gen_unused_slot : for i in 14 to
   63 generate
      rd_data_array(i) <= (others =>
   '0');
   end generate gen_unused_slot;
end arch;
```

Listado 5.6. Subsistema de prueba FPro

```
library ieee;
use ieee.std_logic_1164.all;
use work.chu_io_map.all;
entity mcs_top_sampler is
   generic(BRIDGE_BASE :
   std_logic_vector(31 downto 0) :=
   x"C0000000");
   port(
      clk          : in    std_logic;
      reset_n      : in    std_logic;
      -- switches and LEDs
      sw           : in
   std_logic_vector(15 downto 0);
      led          : out
   std_logic_vector(15 downto 0);
      -- uart
      rx           : in    std_logic;
      tx           : out   std_logic;
      -- xadc
      adc_p        : in
   std_logic_vector(3 downto 0);
      adc_n        : in
   std_logic_vector(3 downto 0);
      -- rgb leds
      rgb_led1     : out
   std_logic_vector(2 downto 0);
      rgb_led2     : out
   std_logic_vector(2 downto 0);
      -- buttons
      btn          : in
   std_logic_vector(4 downto 0);
      -- 4-digit 7-seg LEDs
      an           : out
   std_logic_vector(7 downto 0);
      sseg         : out
   std_logic_vector(7 downto 0);
      -- spi accelerator
--      acl_sclk     : out
   std_logic;
--      acl_mosi     : out
   std_logic;
--      acl_miso     : in
   std_logic;
--      acl_ss_n     : out
   std_logic;
--      -- i2c temperature sensor
--      tmp_i2c_scl : out
   std_logic;
--      tmp_i2c_sda : inout
   std_logic;
--      -- ps2
--      ps2d         : inout
   std_logic;
--      ps2c         : inout
   std_logic;
--      -- nexsys 4 aduio
--      audio_on     : out
   std_logic;
--      audio_pdm    : out
   std_logic;
--      -- PMOD JA (divided into top
   row and bottom row
--      ja_top       : out
   std_logic_vector(4 downto 1);
--      ja_btm       : out
   std_logic_vector(10 downto 7);
      -- to vga monitor
      hsync    : out std_logic;
      vsync    : out std_logic;
      rgb      : out
   std_logic_vector(11 downto 0)

   );
end mcs_top_sampler;

architecture arch of
   mcs_top_sampoler is
   component cpu
      port(
         clk              : in
   std_logic;
         reset            : in
   std_logic;
         io_addr_strobe   : out
   std_logic;
         io_read_strobe   : out
   std_logic;
         io_write_strobe  : out
   std_logic;
         io_address       : out
   std_logic_vector(31 downto 0);
         io_byte_enable   : out
   std_logic_vector(3 downto 0);
         io_write_data    : out
   std_logic_vector(31 downto 0);
         io_read_data     : in
   std_logic_vector(31 downto 0);
         io_ready         : in
   std_logic
      );
   end component;
   component mmcm_fpro
      port(
         clk_in_100M : in
   std_logic;
         clk_100M    : out
   std_logic;
         clk_25M     : out
   std_logic;
         clk_40M     : out
   std_logic;
         clk_67M     : out
   std_logic;
         reset       : in
   std_logic;
         locked      : out std_logic
      );
```

```
    end component;

    signal io_addr_strobe  :
    std_logic;
    signal io_read_strobe  :
    std_logic;
    signal io_write_strobe :
    std_logic;
    signal io_byte_enable  :
    std_logic_vector(3 downto 0);
    signal io_address      :
    std_logic_vector(31 downto 0);
    signal io_write_data   :
    std_logic_vector(31 downto 0);
    signal io_read_data    :
    std_logic_vector(31 downto 0);
    signal io_ready        :
    std_logic;
    signal mmio_cs         :
    std_logic;
    signal mmio_wr         :
    std_logic;
    signal mmio_rd         :
    std_logic;
    signal mmio_addr       :
    std_logic_vector(20 downto 0);
    signal mmio_wr_data    :
    std_logic_vector(31 downto 0);
    signal mmio_rd_data    :
    std_logic_vector(31 downto 0);
    -- clk/reset related
    signal clk_100M        :
    std_logic;
    signal clk_25M         :
    std_logic;
    signal reset_sys       :
    std_logic;
    signal locked          :
    std_logic;
    -- pwm
    signal pwm             :
    std_logic_vector(7 downto 0);
    -- ddfs/audio pdm
    signal pdm             :
    std_logic;
    signal ddfs_sq_wave    :
    std_logic;
    -- fpro bus
    signal fp_wr             :
    std_logic;
    signal fp_addr           :
    std_logic_vector(20 downto 0);
    signal fp_wr_data        :
    std_logic_vector(31 downto 0);
    signal fp_video_cs       :
    std_logic;

begin
    -- clock and reset
--    clk_100M           <= clk;
    -- 100 MHz external clock
--    reset_sys          <= not
    reset_n;
    reset_sys <= (not locked) or (not
    reset_n);
--    -- audio
--    audio_pdm          <= pdm;
--    audio_on           <= '1';
--    --  rgb leds
    rgb_led2           <= pwm(5
    downto 3);
    rgb_led1           <= pwm(2
    downto 0);
--    -- PMOD JA
--    ja_top(1)          <=
    ddfs_sq_wave;
--    ja_top(2)          <= pdm;
--    ja_top(4 downto 3) <= pwm(7
    downto 6);
--    ja_btm             <= "0000";
    -- instantiate clock management
    unit
    clk_mmcm_unit : mmcm_fpro
       port map(
          -- Clock in ports
          clk_in_100M => clk,
          -- Clock out ports
          clk_100M    => clk_100M,
          clk_25M     => clk_25M,
          clk_40M     => open,
          clk_67M     => open,
          -- Status and control
    signals
          reset       => '0',
          locked      => locked
       );
    -- instantiate microBlaze MCS
    mcs_0 : cpu
       port map(
          clk             =>
    clk_100M,
          reset           =>
    reset_sys,
          io_addr_strobe  =>
    io_addr_strobe,
          io_read_strobe  =>
    io_read_strobe,
          io_write_strobe =>
    io_write_strobe,
          io_byte_enable  =>
    io_byte_enable,
          io_address      =>
    io_address,
          io_write_data   =>
    io_write_data,
          io_read_data    =>
    io_read_data,
          io_ready        => io_ready
       );
    -- instantiate MCS IO bus to FPro
    bus bridge
    bridge_unit : entity
    work.chu_mcs_bridge
       generic map(BRG_BASE =>
    BRIDGE_BASE)
       port map(
          io_addr_strobe  =>
    io_addr_strobe,
          io_read_strobe  =>
    io_read_strobe,
```

```
      io_write_strobe =>
io_write_strobe,
      io_byte_enable  =>
io_byte_enable,
      io_address      =>
io_address,
      io_write_data   =>
io_write_data,
      io_read_data    =>
io_read_data,
      io_ready        =>
io_ready,
      fp_video_cs     => open,
      fp_mmio_cs      => mmio_cs,
      fp_wr           => mmio_wr,
      fp_rd           => mmio_rd,
      fp_addr         =>
mmio_addr,
      fp_wr_data      =>
mmio_wr_data,
      fp_rd_data      =>
mmio_rd_data
   );
-- instantiate sampler MMIO
subsystem
mmio_sys_unit : entity
work.mmio_sys_sampler
   port map(
      clk          => clk_100M,
      reset        => reset_sys,
      mmio_cs      => mmio_cs,
      mmio_wr      => mmio_wr,
      mmio_rd      => mmio_rd,
      mmio_addr    => mmio_addr,
      mmio_wr_data =>
   mmio_wr_data,
      mmio_rd_data =>
   mmio_rd_data,
      sw           => sw,
      led          => led,
      rx           => rx,
      tx           => tx,
      adc_p        => adc_p,
      adc_n        => adc_n,
      pwm          => pwm,
      btn          => btn,
      an           => an,
      sseg         => sseg
      tmp_i2c_scl  =>
   tmp_i2c_scl,
      tmp_i2c_sda  =>
   tmp_i2c_sda,
      acl_sclk     => acl_sclk,
      acl_mosi     => acl_mosi,
      acl_miso     => acl_miso,
      acl_ss       => acl_ss_n,
      ps2d         => ps2d,
      ps2c         => ps2c,


      ddfs_sq_wave =>
   ddfs_sq_wave,
      pdm          => pdm
     );
end arch;
```

1.2 Código Software

En este apartado se adjunta el código completo de tipo software generado para la realización de la demo de este proyecto:

Listado Software demo TFG

```
// #define _DEBUG
#include "chu_init.h"
#include "gpio_cores.h"
#include "xadc_core.h"
#include "sseg_core.h"
/*#include "spi_core.h"
#include "i2c_core.h"
#include "ps2_core.h"
#include "ddfs_core.h"
#include "adsr_core.h"
*/
#include "vga_core.h"

/**
 * blink once per second for 5
times.
 * provide a sanity check for timer
(based on SYS_CLK_FREQ)
 * @param led_p pointer to led
instance
 */
void timer_check(GpoCore *led_p) {
   int i;

   for (i = 0; i < 5; i++) {
      led_p->write(0xffff);
      sleep_ms(500);
      led_p->write(0x0000);
      sleep_ms(500);
      debug("timer check - (loop
#)/now: ", i, now_ms());
   }
}

/**
 * check individual led
 * @param led_p pointer to led
instance
 * @param n number of led
 */
void led_check(GpoCore *led_p, int
n) {
   int i, s;

   for (s = 0; s < 2; s++){
      for (i = 0; i < n; i++) {
        led_p->write(1, i);
        sleep_ms(150);
        led_p->write(0, i);
        sleep_ms(150);
      } //for i
   } //for n
}

/**
 * leds flash according to switch
positions.
 * @param led_p pointer to led
instance
 * @param sw_p pointer to switch
instance
 */
void sw_check(GpoCore *led_p,
GpiCore *sw_p) {
   int i, s;

   s = sw_p->read();
   for (i = 0; i < 20; i++) {
      led_p->write(s);
      sleep_ms(300);
      led_p->write(0);
      sleep_ms(300);
   }
}

/**
 * uart transmits test line.
 * @note uart instance is declared
as global variable in chu_io_basic.h
 */
void uart_check() {
   static int loop = 0;

   uart.disp("uart test #");
   uart.disp(loop);
   uart.disp("\n\r");
   loop++;
}

/**
 * read FPGA internal voltage
temperature
 * @param adc_p pointer to xadc
instance
 */

void adc_check(XadcCore *adc_p,
GpoCore *led_p) {
   double reading;
   int n, i;
   uint16_t raw;

   for (i = 0; i < 5; i++) {
      // display 12-bit channel 0
reading in LED
      raw = adc_p->read_raw(0);
      raw = raw >> 4;
      led_p->write(raw);
      // display on-chip sensor and
4 channels in console
      uart.disp("FPGA vcc/temp: ");
```

```
      reading = adc_p-
>read_fpga_vcc();
      uart.disp(reading, 3);
      uart.disp(" / ");
      reading = adc_p-
>read_fpga_temp();
      uart.disp(reading, 3);
      uart.disp("\n\r");
      for (n = 0; n < 4; n++) {
         uart.disp("analog
channel/voltage: ");
         uart.disp(n);
         uart.disp(" / ");
         reading = adc_p-
>read_adc_in(n);
         uart.disp(reading, 3);
         uart.disp("\n\r");
      } // end for
      sleep_ms(200);
   }
}

/**
 * tri-color led dims gradually
 * @param led_p pointer to led
instance
 * @param sw_p pointer to switch
instance
 */

void pwm_3color_led_check(PwmCore
*pwm_p) {
   int i, n;
   double bright, duty;
   const double P20 = 1.2589;  //
P20=100^(1/20); i.e., P20^20=100

   pwm_p->set_freq(50);
   for (n = 0; n < 3; n++) {
      bright = 1.0;
      for (i = 0; i < 20; i++) {
         bright = bright * P20;
         duty = bright / 100.0;
         pwm_p->set_duty(duty, n);
         pwm_p->set_duty(duty, n +
3);

         //pwm_p->set_duty(duty, n +
3);
         sleep_ms(100);
      }
      sleep_ms(300);
      pwm_p->set_duty(0.0, n);
      pwm_p->set_duty(0.0, n + 3);
   }
}

/**
 * Test debounced buttons
 *   - count transitions of normal
and debounced button
 * @param db_p pointer to
debouceCore instance
 */
void debounce_check(DebounceCore
*db_p, GpoCore *led_p) {
   long start_time;
   int btn_old, db_old, btn_new,
db_new;
   int b = 0;
   int d = 0;
   uint32_t ptn;

   start_time = now_ms();
   btn_old = db_p->read();
   db_old = db_p->read_db();
   do {
      btn_new = db_p->read();
      db_new = db_p->read_db();
      if (btn_old != btn_new) {
         b = b + 1;
         btn_old = btn_new;
      }
      if (db_old != db_new) {
         d = d + 1;
         db_old = db_new;
      }
      ptn = d & 0x0000000f;
      ptn = ptn | (b & 0x0000000f)
<< 4;
      led_p->write(ptn);
   } while ((now_ms() - start_time)
< 5000);
}

/**
 * Test pattern in 7-segment LEDs
 * @param sseg_p pointer to 7-seg
LED instance
 */

void sseg_check(SsegCore *sseg_p) {
   int i, n, s, m;
   uint8_t dp;

   //turn off led
   for (i = 0; i < 8; i++) {
      sseg_p->write_1ptn(0xff, i);
   }
   //turn off all decimal points
   sseg_p->set_dp(0x00);

   // display 0x0 to 0xf in 4 epochs
   // upper 4  digits mirror the
lower 4
   //for (n = 0; n < 16; n++) {
     //for (i = 0; i < 4; i++) {
      //for ((i = 0; i < 8; i++) &
(n = 0; n < 16; n++)) {
      for (s = 0; s < 3; s++) {
        n = 0;
        i = 0;
        m = 200;
          //--sseg_p-
>write_1ptn(sseg_p->h2s(i + n * 4),
3 - i);
         //--sseg_p-
>write_1ptn(sseg_p->h2s(i + n * 4),
7 - i);
```

```
        sseg_p->write_1ptn(sseg_p-
>h2s( n ), i );
        sleep_ms(m);
        sseg_p->write_1ptn(sseg_p-
>h2s( n ), i+1 );
        sseg_p->write_1ptn(sseg_p-
>h2s( n+1 ), i );
        sleep_ms(m);
        sseg_p->write_1ptn(sseg_p-
>h2s( n ), i+2 );
        sseg_p->write_1ptn(sseg_p-
>h2s( n+1 ), i+1 );
        sseg_p->write_1ptn(sseg_p-
>h2s( n+2 ), i );
        sleep_ms(m);
        sseg_p->write_1ptn(sseg_p-
>h2s( n ), i+3 );
        sseg_p->write_1ptn(sseg_p-
>h2s( n+1 ), i+2 );
        sseg_p->write_1ptn(sseg_p-
>h2s( n+2 ), i+1 );
        sseg_p->write_1ptn(sseg_p-
>h2s( n+16 ), i );
        sleep_ms(m);
        sseg_p->write_1ptn(sseg_p-
>h2s( n ), i+4 );
        sseg_p->write_1ptn(sseg_p-
>h2s( n+1 ), i+3 );
        sseg_p->write_1ptn(sseg_p-
>h2s( n+2 ), i+2 );
        sseg_p->write_1ptn(sseg_p-
>h2s( n+16 ), i+1 );
        sseg_p->write_1ptn(sseg_p-
>h2s( n+3 ), i );
        sleep_ms(m);
        sseg_p->write_1ptn(sseg_p-
>h2s( n ), i+5 );
        sseg_p->write_1ptn(sseg_p-
>h2s( n+1 ), i+4 );
        sseg_p->write_1ptn(sseg_p-
>h2s( n+2 ), i+3 );
        sseg_p->write_1ptn(sseg_p-
>h2s( n+16 ), i+2 );
        sseg_p->write_1ptn(sseg_p-
>h2s( n+3 ), i+1 );
        sseg_p->write_1ptn(sseg_p-
>h2s( n+4 ), i );
        sleep_ms(m);
        sseg_p->write_1ptn(sseg_p-
>h2s( n ), i+6 );
        sseg_p->write_1ptn(sseg_p-
>h2s( n+1 ), i+5 );
        sseg_p->write_1ptn(sseg_p-
>h2s( n+2 ), i+4 );
        sseg_p->write_1ptn(sseg_p-
>h2s( n+16 ), i+3 );
        sseg_p->write_1ptn(sseg_p-
>h2s( n+3 ), i+2 );
        sseg_p->write_1ptn(sseg_p-
>h2s( n+4 ), i+1 );
        sseg_p->write_1ptn(sseg_p-
>h2s( n+5 ), i );
        sleep_ms(m);
        sseg_p->write_1ptn(sseg_p-
>h2s( n ), i+7 );
        sseg_p->write_1ptn(sseg_p-
>h2s( n+1 ), i+6 );
        sseg_p->write_1ptn(sseg_p-
>h2s( n+2 ), i+5 );
        sseg_p->write_1ptn(sseg_p-
>h2s( n+16 ), i+4 );
        sseg_p->write_1ptn(sseg_p-
>h2s( n+3 ), i+3 );
        sseg_p->write_1ptn(sseg_p-
>h2s( n+4 ), i+2 );
        sseg_p->write_1ptn(sseg_p-
>h2s( n+5 ), i+1 );
        sseg_p->write_1ptn(sseg_p-
>h2s( n+6 ), i );
        sleep_ms(m);
        sseg_p->write_1ptn(sseg_p-
>h2s( n+1 ), i+7 );
        sseg_p->write_1ptn(sseg_p-
>h2s( n+2 ), i+6 );
        sseg_p->write_1ptn(sseg_p-
>h2s( n+16 ), i+5 );
        sseg_p->write_1ptn(sseg_p-
>h2s( n+3 ), i+4 );
        sseg_p->write_1ptn(sseg_p-
>h2s( n+4 ), i+3 );
        sseg_p->write_1ptn(sseg_p-
>h2s( n+5 ), i+2 );
        sseg_p->write_1ptn(sseg_p-
>h2s( n+6 ), i+1 );
        sseg_p->write_1ptn(sseg_p-
>h2s( n+7 ), i );
        sleep_ms(m);
        sseg_p->write_1ptn(sseg_p-
>h2s( n+2 ), i+7 );
        sseg_p->write_1ptn(sseg_p-
>h2s( n+16 ), i+6 );
        sseg_p->write_1ptn(sseg_p-
>h2s( n+3 ), i+5 );
        sseg_p->write_1ptn(sseg_p-
>h2s( n+4 ), i+4 );
        sseg_p->write_1ptn(sseg_p-
>h2s( n+5 ), i+3 );
        sseg_p->write_1ptn(sseg_p-
>h2s( n+6 ), i+2 );
        sseg_p->write_1ptn(sseg_p-
>h2s( n+7 ), i+1 );
        sseg_p->write_1ptn(sseg_p-
>h2s( n+8 ), i );
        sleep_ms(m);
        sseg_p->write_1ptn(sseg_p-
>h2s( n+16 ), i+7 );
        sseg_p->write_1ptn(sseg_p-
>h2s( n+3 ), i+6 );
        sseg_p->write_1ptn(sseg_p-
>h2s( n+4 ), i+5 );
        sseg_p->write_1ptn(sseg_p-
>h2s( n+5 ), i+4 );
        sseg_p->write_1ptn(sseg_p-
>h2s( n+6 ), i+3 );
        sseg_p->write_1ptn(sseg_p-
>h2s( n+7 ), i+2 );
        sseg_p->write_1ptn(sseg_p-
>h2s( n+8 ), i+1 );
        sseg_p->write_1ptn(sseg_p-
>h2s( n+16 ), i );
```

```
        sleep_ms(m);
        sseg_p->write_1ptn(sseg_p-
>h2s( n+3 ), i+7 );
        sseg_p->write_1ptn(sseg_p-
>h2s( n+4 ), i+6 );
        sseg_p->write_1ptn(sseg_p-
>h2s( n+5 ), i+5 );
        sseg_p->write_1ptn(sseg_p-
>h2s( n+6 ), i+4 );
        sseg_p->write_1ptn(sseg_p-
>h2s( n+7 ), i+3 );
        sseg_p->write_1ptn(sseg_p-
>h2s( n+8 ), i+2 );
        sseg_p->write_1ptn(sseg_p-
>h2s( n+16 ), i+1 );
        sseg_p->write_1ptn(sseg_p-
>h2s( n+9 ), i );
        sleep_ms(m);
        sseg_p->write_1ptn(sseg_p-
>h2s( n+4 ), i+7 );
        sseg_p->write_1ptn(sseg_p-
>h2s( n+5 ), i+6 );
        sseg_p->write_1ptn(sseg_p-
>h2s( n+6 ), i+5 );
        sseg_p->write_1ptn(sseg_p-
>h2s( n+7 ), i+4 );
        sseg_p->write_1ptn(sseg_p-
>h2s( n+8 ), i+3 );
        sseg_p->write_1ptn(sseg_p-
>h2s( n+16 ), i+2 );
        sseg_p->write_1ptn(sseg_p-
>h2s( n+9 ), i+1 );
        sseg_p->write_1ptn(sseg_p-
>h2s( n+10 ), i );
        sleep_ms(m);
        sseg_p->write_1ptn(sseg_p-
>h2s( n+5 ), i+7 );
        sseg_p->write_1ptn(sseg_p-
>h2s( n+6 ), i+6 );
        sseg_p->write_1ptn(sseg_p-
>h2s( n+7 ), i+5 );
        sseg_p->write_1ptn(sseg_p-
>h2s( n+8 ), i+4 );
        sseg_p->write_1ptn(sseg_p-
>h2s( n+16 ), i+3 );
        sseg_p->write_1ptn(sseg_p-
>h2s( n+9 ), i+2 );
        sseg_p->write_1ptn(sseg_p-
>h2s( n+10 ), i+1 );
        sseg_p->write_1ptn(sseg_p-
>h2s( n+11 ), i );
        sleep_ms(m);
        sseg_p->write_1ptn(sseg_p-
>h2s( n+6 ), i+7 );
        sseg_p->write_1ptn(sseg_p-
>h2s( n+7 ), i+6 );
        sseg_p->write_1ptn(sseg_p-
>h2s( n+8 ), i+5 );
        sseg_p->write_1ptn(sseg_p-
>h2s( n+16 ), i+4 );
        sseg_p->write_1ptn(sseg_p-
>h2s( n+9 ), i+3 );
        sseg_p->write_1ptn(sseg_p-
>h2s( n+10 ), i+2 );
        sseg_p->write_1ptn(sseg_p-
>h2s( n+11 ), i+1 );
        sseg_p->write_1ptn(sseg_p-
>h2s( n+12 ), i );
        sleep_ms(m);
        sseg_p->write_1ptn(sseg_p-
>h2s( n+7 ), i+7 );
        sseg_p->write_1ptn(sseg_p-
>h2s( n+8 ), i+6 );
        sseg_p->write_1ptn(sseg_p-
>h2s( n+16 ), i+5 );
        sseg_p->write_1ptn(sseg_p-
>h2s( n+9 ), i+4 );
        sseg_p->write_1ptn(sseg_p-
>h2s( n+10 ), i+3 );
        sseg_p->write_1ptn(sseg_p-
>h2s( n+11 ), i+2 );
        sseg_p->write_1ptn(sseg_p-
>h2s( n+12 ), i+1 );
        sseg_p->write_1ptn(sseg_p-
>h2s( n+13 ), i );
        sleep_ms(m);
        sseg_p->write_1ptn(sseg_p-
>h2s( n+8 ), i+7 );
        sseg_p->write_1ptn(sseg_p-
>h2s( n+16 ), i+6 );
        sseg_p->write_1ptn(sseg_p-
>h2s( n+9 ), i+5 );
        sseg_p->write_1ptn(sseg_p-
>h2s( n+10 ), i+4 );
        sseg_p->write_1ptn(sseg_p-
>h2s( n+11 ), i+3 );
        sseg_p->write_1ptn(sseg_p-
>h2s( n+12 ), i+2 );
        sseg_p->write_1ptn(sseg_p-
>h2s( n+13 ), i+1 );
        sseg_p->write_1ptn(sseg_p-
>h2s( n+14 ), i );
        sleep_ms(m);
        sseg_p->write_1ptn(sseg_p-
>h2s( n+16 ), i+7 );
        sseg_p->write_1ptn(sseg_p-
>h2s( n+9 ), i+6 );
        sseg_p->write_1ptn(sseg_p-
>h2s( n+10 ), i+5 );
        sseg_p->write_1ptn(sseg_p-
>h2s( n+11 ), i+4 );
        sseg_p->write_1ptn(sseg_p-
>h2s( n+12 ), i+3 );
        sseg_p->write_1ptn(sseg_p-
>h2s( n+13 ), i+2 );
        sseg_p->write_1ptn(sseg_p-
>h2s( n+14 ), i+1 );
        sseg_p->write_1ptn(sseg_p-
>h2s( n+15 ), i );
        sleep_ms(m);
        sseg_p->write_1ptn(sseg_p-
>h2s( n+9 ), i+7 );
        sseg_p->write_1ptn(sseg_p-
>h2s( n+10 ), i+6 );
        sseg_p->write_1ptn(sseg_p-
>h2s( n+11 ), i+5 );
        sseg_p->write_1ptn(sseg_p-
>h2s( n+12 ), i+4 );
```

```
        sseg_p->write_1ptn(sseg_p-
>h2s( n+13 ), i+3 );
        sseg_p->write_1ptn(sseg_p-
>h2s( n+14 ), i+2 );
        sseg_p->write_1ptn(sseg_p-
>h2s( n+15 ), i+1 );
        sseg_p->write_1ptn(sseg_p-
>h2s( n+16 ), i );
        sleep_ms(m);
        sseg_p->write_1ptn(sseg_p-
>h2s( n+10 ), i+7 );
        sseg_p->write_1ptn(sseg_p-
>h2s( n+11 ), i+6 );
        sseg_p->write_1ptn(sseg_p-
>h2s( n+12 ), i+5 );
        sseg_p->write_1ptn(sseg_p-
>h2s( n+13 ), i+4 );
        sseg_p->write_1ptn(sseg_p-
>h2s( n+14 ), i+3 );
        sseg_p->write_1ptn(sseg_p-
>h2s( n+15 ), i+2 );
        sseg_p->write_1ptn(sseg_p-
>h2s( n+16 ), i+1 );
        sseg_p->write_1ptn(sseg_p-
>h2s( n+16 ), i );
        sleep_ms(m);
        sseg_p->write_1ptn(sseg_p-
>h2s( n+11 ), i+7 );
        sseg_p->write_1ptn(sseg_p-
>h2s( n+12 ), i+6 );
        sseg_p->write_1ptn(sseg_p-
>h2s( n+13 ), i+5 );
        sseg_p->write_1ptn(sseg_p-
>h2s( n+14 ), i+4 );
        sseg_p->write_1ptn(sseg_p-
>h2s( n+15 ), i+3 );
        sseg_p->write_1ptn(sseg_p-
>h2s( n+16 ), i+2 );
        sseg_p->write_1ptn(sseg_p-
>h2s( n+16 ), i+1 );
        sseg_p->write_1ptn(sseg_p-
>h2s( n+16 ), i );
        sleep_ms(m);
        sseg_p->write_1ptn(sseg_p-
>h2s( n+12 ), i+7 );
        sseg_p->write_1ptn(sseg_p-
>h2s( n+13 ), i+6 );
        sseg_p->write_1ptn(sseg_p-
>h2s( n+14 ), i+5 );
        sseg_p->write_1ptn(sseg_p-
>h2s( n+15 ), i+4 );
        sseg_p->write_1ptn(sseg_p-
>h2s( n+16 ), i+3 );
        sseg_p->write_1ptn(sseg_p-
>h2s( n+16 ), i+2 );
        sseg_p->write_1ptn(sseg_p-
>h2s( n+16 ), i+1 );
        sseg_p->write_1ptn(sseg_p-
>h2s( n+16 ), i );
        sleep_ms(m);
        sseg_p->write_1ptn(sseg_p-
>h2s( n+13 ), i+7 );
        sseg_p->write_1ptn(sseg_p-
>h2s( n+14 ), i+6 );
        sseg_p->write_1ptn(sseg_p-
>h2s( n+15 ), i+5 );
        sseg_p->write_1ptn(sseg_p-
>h2s( n+16 ), i+4 );
        sseg_p->write_1ptn(sseg_p-
>h2s( n+16 ), i+3 );
        sseg_p->write_1ptn(sseg_p-
>h2s( n+16 ), i+2 );
        sseg_p->write_1ptn(sseg_p-
>h2s( n+16 ), i+1 );
        sseg_p->write_1ptn(sseg_p-
>h2s( n+16 ), i );
        sleep_ms(m);
        sseg_p->write_1ptn(sseg_p-
>h2s( n+14 ), i+7 );
        sseg_p->write_1ptn(sseg_p-
>h2s( n+15 ), i+6 );
        sseg_p->write_1ptn(sseg_p-
>h2s( n+16 ), i+5 );
        sseg_p->write_1ptn(sseg_p-
>h2s( n+16 ), i+4 );
        sseg_p->write_1ptn(sseg_p-
>h2s( n+16 ), i+3 );
        sseg_p->write_1ptn(sseg_p-
>h2s( n+16 ), i+2 );
        sseg_p->write_1ptn(sseg_p-
>h2s( n+16 ), i+1 );
        sseg_p->write_1ptn(sseg_p-
>h2s( n+16 ), i );
        sleep_ms(m);
        sseg_p->write_1ptn(sseg_p-
>h2s( n+15 ), i+7 );
        sseg_p->write_1ptn(sseg_p-
>h2s( n+16 ), i+6 );
        sseg_p->write_1ptn(sseg_p-
>h2s( n+16 ), i+5 );
        sseg_p->write_1ptn(sseg_p-
>h2s( n+16 ), i+4 );
        sseg_p->write_1ptn(sseg_p-
>h2s( n+16 ), i+3 );
        sseg_p->write_1ptn(sseg_p-
>h2s( n+16 ), i+2 );
        sseg_p->write_1ptn(sseg_p-
>h2s( n+16 ), i+1 );
        sseg_p->write_1ptn(sseg_p-
>h2s( n+16 ), i );
        sleep_ms(m);
        sseg_p->write_1ptn(sseg_p-
>h2s( n+16 ), i+7 );
        sseg_p->write_1ptn(sseg_p-
>h2s( n+16 ), i+6 );
        sseg_p->write_1ptn(sseg_p-
>h2s( n+16 ), i+5 );
        sseg_p->write_1ptn(sseg_p-
>h2s( n+16 ), i+4 );
        sseg_p->write_1ptn(sseg_p-
>h2s( n+16 ), i+3 );
        sseg_p->write_1ptn(sseg_p-
>h2s( n+16 ), i+2 );
        sseg_p->write_1ptn(sseg_p-
>h2s( n+16 ), i+1 );
        sseg_p->write_1ptn(sseg_p-
>h2s( n+16 ), i );
        sleep_ms(m);
```

```
      } // for i
   //}  // for n

      // shift a decimal point 4
times
  for (i = 0; i < 8; i++) {
      bit_set(dp, 7 - i);
      sseg_p->set_dp(1 << (7 - i));
      sleep_ms(300);
   }
   //turn off led
   for (i = 0; i < 8; i++) {
      sseg_p->write_1ptn(0xff, i);
   }
   //turn off all decimal points
   sseg_p->set_dp(0x00);


}

/*
 *
 * Test adxl362 accelerometer using
SPI


void gsensor_check(SpiCore *spi_p,
GpoCore *led_p) {
   const uint8_t RD_CMD = 0x0b;
   const uint8_t PART_ID_REG = 0x02;
   const uint8_t DATA_REG = 0x08;
   const float raw_max = 127.0 /
2.0;  //128 max 8-bit reading for
+/-2g

   int8_t xraw, yraw, zraw;
   float x, y, z;
   int id;

   spi_p->set_freq(400000);
   spi_p->set_mode(0, 0);
   // check part id
   spi_p->assert_ss(0);    //
activate
   spi_p->transfer(RD_CMD);  // for
read operation
   spi_p->transfer(PART_ID_REG);  //
part id address
   id = (int) spi_p->transfer(0x00);
   spi_p->deassert_ss(0);
   uart.disp("read ADXL362 id
(should be 0xf2): ");
   uart.disp(id, 16);
   uart.disp("\n\r");
   // read 8-bit x/y/z g values once
   spi_p->assert_ss(0);    //
activate
   spi_p->transfer(RD_CMD);  // for
read operation
   spi_p->transfer(DATA_REG);  //
   xraw = spi_p->transfer(0x00);
   yraw = spi_p->transfer(0x00);
   zraw = spi_p->transfer(0x00);
   spi_p->deassert_ss(0);
   x = (float) xraw / raw_max;
   y = (float) yraw / raw_max;
   z = (float) zraw / raw_max;
   uart.disp("x/y/z axis g values:
");
   uart.disp(x, 3);
   uart.disp(" / ");
   uart.disp(y, 3);
   uart.disp(" / ");
   uart.disp(z, 3);
   uart.disp("\n\r");
}

 *
 * read temperature from adt7420
 * @param adt7420_p pointer to
adt7420 instance

void adt7420_check(I2cCore
*adt7420_p, GpoCore *led_p) {
   const uint8_t DEV_ADDR = 0x4b;
   uint8_t wbytes[2], bytes[2];
   //int ack;
   uint16_t tmp;
   float tmpC;

   // read adt7420 id register to
verify device existence
   // ack = adt7420_p-
>read_dev_reg_byte(DEV_ADDR, 0x0b,
&id);

   wbytes[0] = 0x0b;
   adt7420_p-
>write_transaction(DEV_ADDR, wbytes,
1, 1);
   adt7420_p-
>read_transaction(DEV_ADDR, bytes,
1, 0);
   uart.disp("read ADT7420 id
(should be 0xcb): ");
   uart.disp(bytes[0], 16);
   uart.disp("\n\r");
   //debug("ADT check ack/id: ",
ack, bytes[0]);
   // read 2 bytes
   //ack = adt7420_p-
>read_dev_reg_bytes(DEV_ADDR, 0x0,
bytes, 2);
   wbytes[0] = 0x00;
   adt7420_p-
>write_transaction(DEV_ADDR, wbytes,
1, 1);
   adt7420_p-
>read_transaction(DEV_ADDR, bytes,
2, 0);

   // conversion
   tmp = (uint16_t) bytes[0];
   tmp = (tmp << 8) + (uint16_t)
bytes[1];
   if (tmp & 0x8000) {
      tmp = tmp >> 3;
      tmpC = (float) ((int) tmp -
8192) / 16;
   } else {
```

```
      tmp = tmp >> 3;
      tmpC = (float) tmp / 16;
   }
   uart.disp("temperature (C): ");
   uart.disp(tmpC);
   uart.disp("\n\r");
   led_p->write(tmp);
   sleep_ms(1000);
   led_p->write(0);
}

void ps2_check(Ps2Core *ps2_p) {
   int id;
   int lbtn, rbtn, xmov, ymov;
   char ch;
   unsigned long last;

   uart.disp("\n\rPS2 device (1-
keyboard / 2-mouse): ");
   id = ps2_p->init();
   uart.disp(id);
   uart.disp("\n\r");
   last = now_ms();
   do {
      if (id == 2) {  // mouse
         if (ps2_p-
>get_mouse_activity(&lbtn, &rbtn,
&xmov, &ymov)) {
            uart.disp("[");
            uart.disp(lbtn);
            uart.disp(", ");
            uart.disp(rbtn);
            uart.disp(", ");
            uart.disp(xmov);
            uart.disp(", ");
            uart.disp(ymov);
            uart.disp("] \r\n");
            last = now_ms();

         }   // end
get_mouse_activitiy()
      } else {
         if (ps2_p->get_kb_ch(&ch))
{
            uart.disp(ch);
            uart.disp(" ");
            last = now_ms();
         } // end get_kb_ch()
      }  // end id==2
   } while (now_ms() - last < 5000);
   uart.disp("\n\rExit PS2 test
\n\r");

}


 * play primary notes with ddfs
 * @param ddfs_p pointer to ddfs
core
 * @note: music tempo is defined as
beats of quarter-note per minute.
 *        60 bpm is 1 sec per
quarter note
 * @note "click" sound due to abrupt
stop of a note
 *

void ddfs_check(DdfsCore *ddfs_p,
GpoCore *led_p) {
   int i, j;
   float env;

   //vol =
(float)sw.read_pin()/(float)(1<<16),
   ddfs_p->set_env_source(0);  //
select envelop source
   ddfs_p->set_env(0.0);   // set
volume
   sleep_ms(500);
   ddfs_p->set_env(1.0);   // set
volume
   ddfs_p->set_carrier_freq(262);
   sleep_ms(2000);
   ddfs_p->set_env(0.0);   // set
volume
   sleep_ms(2000);
   // volume control (attenuation)
   ddfs_p->set_env(0.0);   // set
volume
   env = 1.0;
   for (i = 0; i < 1000; i++) {
      ddfs_p->set_env(env);
      sleep_ms(10);
      env = env / 1.0109;
//1.0109**1024=2**16
   }
   // frequency modulation 635-912
800 - 2000 siren sound
   ddfs_p->set_env(1.0);   // set
volume
   ddfs_p->set_carrier_freq(635);
   for (i = 0; i < 5; i++) {
// 10 cycles
      for (j = 0; j < 30; j++) {
// sweep 30 steps
         ddfs_p->set_offset_freq(j *
10);  // 10 Hz increment
         sleep_ms(25);
      } // end j loop
   } // end i loop
   ddfs_p->set_offset_freq(0);
   ddfs_p->set_env(0.0);   // set
volume
   sleep_ms(1000);
}


 * play primary notes with ddfs
 * @param adsr_p pointer to adsr
core
 * @param ddfs_p pointer to ddfs
core
 * @note: music tempo is defined as
beats of quarter-note per minute.
 *        60 bpm is 1 sec per
quarter note
 *

void adsr_check(AdsrCore *adsr_p,
GpoCore *led_p, GpiCore *sw_p) {
```

```
   const int melody[] = { 0, 2, 4,
5, 7, 9, 11 };
   int i, oct;

   adsr_p->init();
   // no adsr envelop and  play one
octave
   adsr_p->bypass();
   for (i = 0; i < 7; i++) {
      led_p->write(bit(i));
      adsr_p->play_note(melody[i],
3, 500);
      sleep_ms(500);
   }
   adsr_p->abort();
   sleep_ms(1000);
   // set and enable adsr envelop
   // play 4 octaves
   adsr_p->select_env(sw_p->read());
   for (oct = 3; oct < 6; oct++) {
      for (i = 0; i < 7; i++) {
         led_p->write(bit(i));
         adsr_p-
>play_note(melody[i], oct, 500);
         sleep_ms(500);
      }
   }
   led_p->write(0);
   // test duration
   sleep_ms(1000);
   for (i = 0; i < 4; i++) {
      adsr_p->play_note(0, 4, 500 *
i);
      sleep_ms(500 * i + 1000);
   }
}


 * check bar generator core
 * @param bar_p pointer to Gpv
instance
 */
void bar_check(GpvCore *bar_p) {
   bar_p->bypass(0);
   sleep_ms(3000);
}

/**
 * check color-to-grayscale core
 * @param gray_p pointer to Gpv
instance
 */
void gray_check(GpvCore *gray_p) {
   gray_p->bypass(0);
   sleep_ms(3000);
   gray_p->bypass(1);
}

/**
 * check osd core
 * @param osd_p pointer to osd
instance
 */
void osd_check(OsdCore *osd_p) {
   osd_p->set_color(0x0f0, 0x001);
// dark gray/green
   osd_p->bypass(0);
   osd_p->clr_screen();
   for (int i = 0; i < 64; i++) {
      osd_p->wr_char(8 + i, 20, i);
      osd_p->wr_char(8 + i, 21, 64 +
i, 1);
      sleep_ms(100);
   }
   sleep_ms(3000);
}

/**
 * test frame buffer core
 * @param frame_p pointer to frame
buffer instance
 */
void frame_check(FrameCore *frame_p)
{
   int x, y, color;

   frame_p->bypass(0);
   for (int i = 0; i < 10; i++) {
      frame_p->clr_screen(0x008);
// dark green
      for (int j = 0; j < 20; j++) {
         x = rand() % 640;
         y = rand() % 480;
         color = rand() % 512;
         frame_p->plot_line(400,
200, x, y, color);
      }
      sleep_ms(300);
   }
   sleep_ms(3000);
}

/**
 * test ghost sprite
 * @param ghost_p pointer to mouse
sprite instance
 */
void mouse_check(SpriteCore
*mouse_p) {
   int x, y;

   mouse_p->bypass(0);
   // clear top and bottom lines
   for (int i = 0; i < 32; i++) {
      mouse_p->wr_mem(i, 0);
      mouse_p->wr_mem(31 * 32 + i,
0);
   }

   // slowly move mouse pointer
   x = 0;
   y = 0;
   for (int i = 0; i < 80; i++) {
      mouse_p->move_xy(x, y);
      sleep_ms(50);
      x = x + 4;
      y = y + 3;
   }
   sleep_ms(3000);
```

```
   // load top and bottom rows
   for (int i = 0; i < 32; i++) {
      sleep_ms(20);
      mouse_p->wr_mem(i, 0x00f);
      mouse_p->wr_mem(31 * 32 + i,
0xf00);
   }
   sleep_ms(3000);
}

/**
 * test ghost sprite
 * @param ghost_p pointer to ghost
sprite instance
 */
void ghost_check(SpriteCore
*ghost_p) {
   int x, y;

   // slowly move mouse pointer
   ghost_p->bypass(0);
   ghost_p->wr_ctrl(0x1c);
//animation; blue ghost
   x = 0;
   y = 100;
   for (int i = 0; i < 156; i++) {
      ghost_p->move_xy(x, y);
      sleep_ms(100);
      x = x + 4;
      if (i == 80) {
         // change to red ghost half
way
         ghost_p->wr_ctrl(0x04);
      }
   }
   sleep_ms(3000);
}

/**
 * core test
 * @param led_p pointer to led
instance
 * @param sw_p pointer to switch
instance
 */
void show_test_id(int n, GpoCore
*led_p) {
   int i, ptn;

   ptn = n; //1 << n;
   for (i = 0; i < 20; i++) {
      led_p->write(ptn);
      sleep_ms(30);
      led_p->write(0);
      sleep_ms(30);
   }
}

GpoCore
led(get_slot_addr(BRIDGE_BASE,
S2_LED));
GpiCore
sw(get_slot_addr(BRIDGE_BASE,
S3_SW));
XadcCore
adc(get_slot_addr(BRIDGE_BASE,
S5_XDAC));
PwmCore
pwm(get_slot_addr(BRIDGE_BASE,
S6_PWM));
DebounceCore
btn(get_slot_addr(BRIDGE_BASE,
S7_BTN));
SsegCore
sseg(get_slot_addr(BRIDGE_BASE,
S8_SSEG));
/*SpiCore
spi(get_slot_addr(BRIDGE_BASE,
S9_SPI));
I2cCore
adt7420(get_slot_addr(BRIDGE_BASE,
S10_I2C));
Ps2Core
ps2(get_slot_addr(BRIDGE_BASE,
S11_PS2));
DdfsCore
ddfs(get_slot_addr(BRIDGE_BASE,
S12_DDFS));
AdsrCore
adsr(get_slot_addr(BRIDGE_BASE,
S13_ADSR), &ddfs);
*/
// video cores
FrameCore frame(FRAME_BASE);
GpvCore
bar(get_sprite_addr(BRIDGE_BASE,
V7_BAR));
GpvCore
gray(get_sprite_addr(BRIDGE_BASE,
V6_GRAY));
SpriteCore
ghost(get_sprite_addr(BRIDGE_BASE,
V3_GHOST), 1024);
SpriteCore
mouse(get_sprite_addr(BRIDGE_BASE,
V1_MOUSE), 1024);
OsdCore
osd(get_sprite_addr(BRIDGE_BASE,
V2_OSD));


int main() {
   //uint8_t id, ;

   timer_check(&led);
   while (1) {
     sleep_ms(1000);
      show_test_id(1, &led);
      led_check(&led, 16);
     sleep_ms(1000);
      sw_check(&led, &sw);
     sleep_ms(1000);
      show_test_id(3, &led);
      uart_check();
     sleep_ms(1000);
      debug("main - switch value /
up time : ", sw.read(), now_ms());
     sleep_ms(1000);
      show_test_id(5, &led);
```

```
  adc_check(&adc, &led);
 sleep_ms(1000);
  show_test_id(6, &led);
  pwm_3color_led_check(&pwm);
 sleep_ms(1000);
  show_test_id(7, &led);
  debounce_check(&btn, &led);
 sleep_ms(1000);
  show_test_id(8, &led);
  sseg_check(&sseg);
 sleep_ms(1000);
  show_test_id(9, &led);
  /*gsensor_check(&spi, &led);
  show_test_id(10, &led);
  adt7420_check(&adt7420, &led);
  show_test_id(11, &led);
  ps2_check(&ps2);
  show_test_id(12, &led);
  ddfs_check(&ddfs, &led);
  show_test_id(13, &led);
  adsr_check(&adsr, &led, &sw);
 */
  // video test
        /*show_test_id(14, &led);
        frame.bypass(1);
        bar.bypass(1);
        gray.bypass(1);
        ghost.bypass(1);
        osd.bypass(1);
        mouse.bypass(1);
        // enable video cores one by
one
        frame_check(&frame);
        bar_check(&bar);
        gray_check(&gray);
        ghost_check(&ghost);
        osd_check(&osd);
        mouse_check(&mouse);*/
    } //while
      sleep_ms(2000);
      timer_check(&led);
 } //main
```

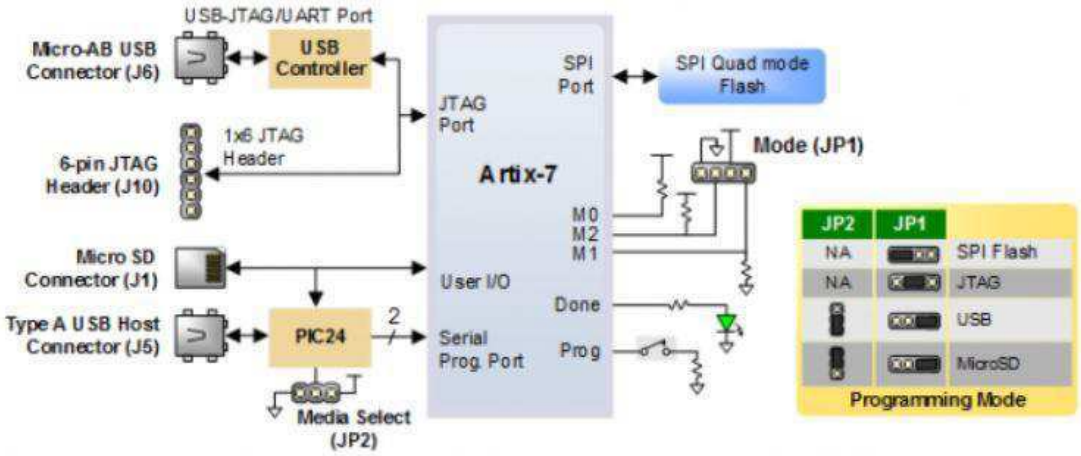
ESQUEMAS

Índice

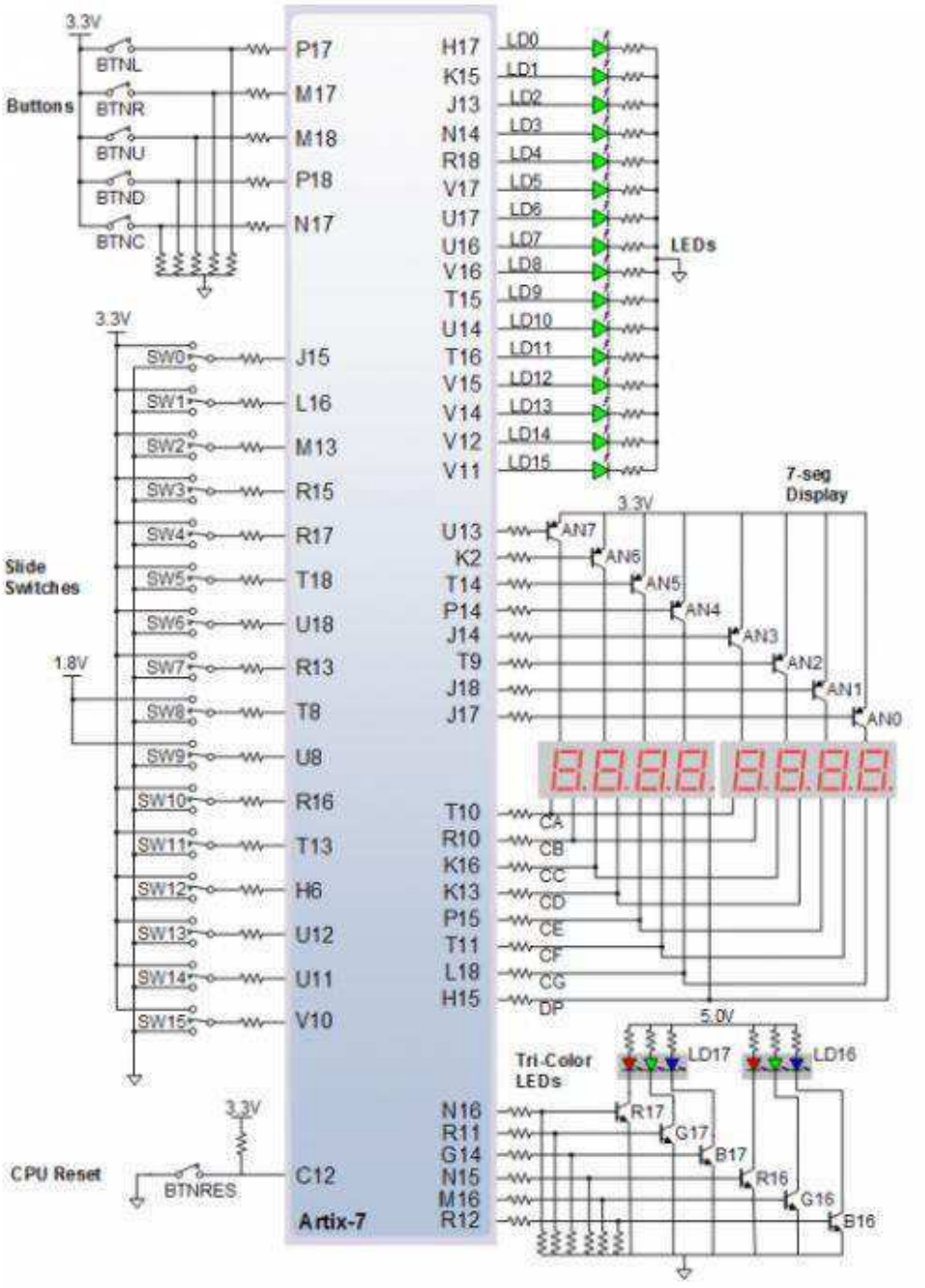
1. Documentación gráfica

Estableciendo esta sección de esquemas como una sección de documentación gráfica, se incorporan a continuación los siguientes gráficos o esquemas referidos los distintos componentes de los que está compuesta una tarjeta Nexys 4 DDR.

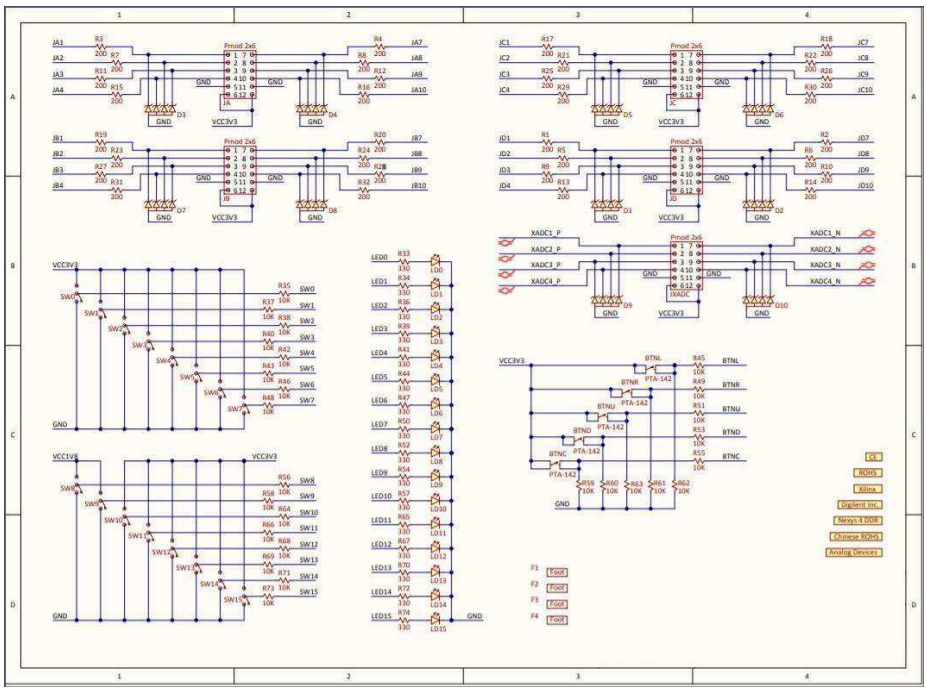
1.1 Configuración FPGA



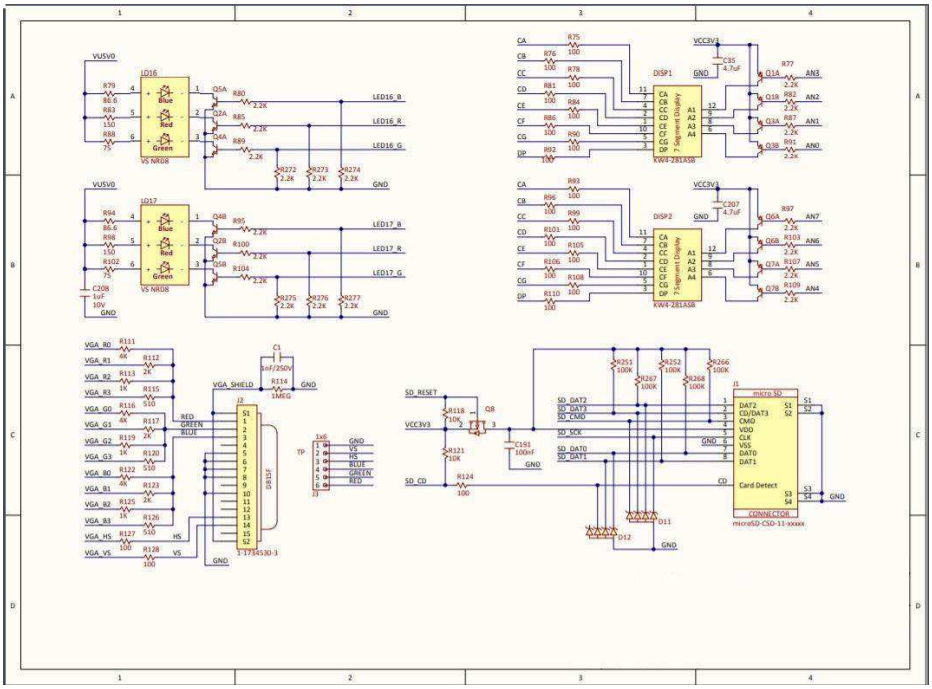
1.2 Conexiones de E/S básicas



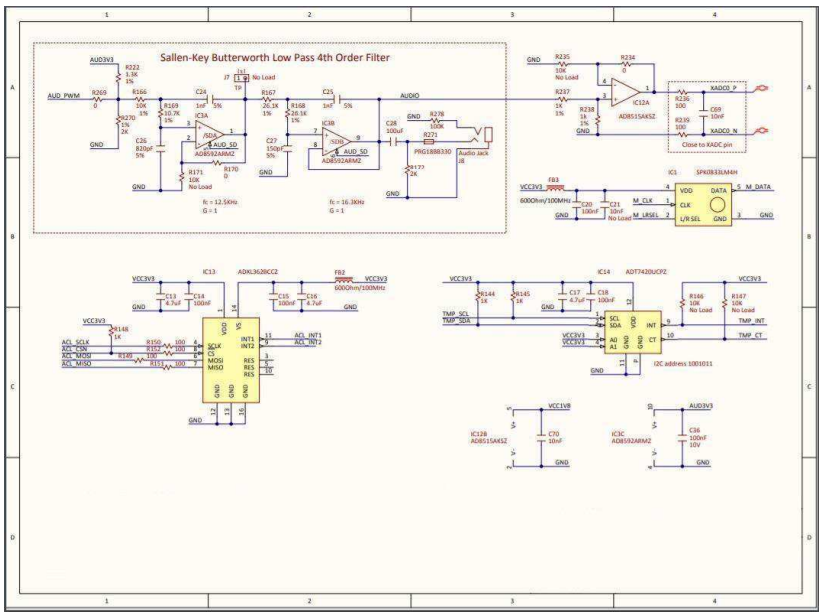
1.3 Circuito de E/S Pmod OLED



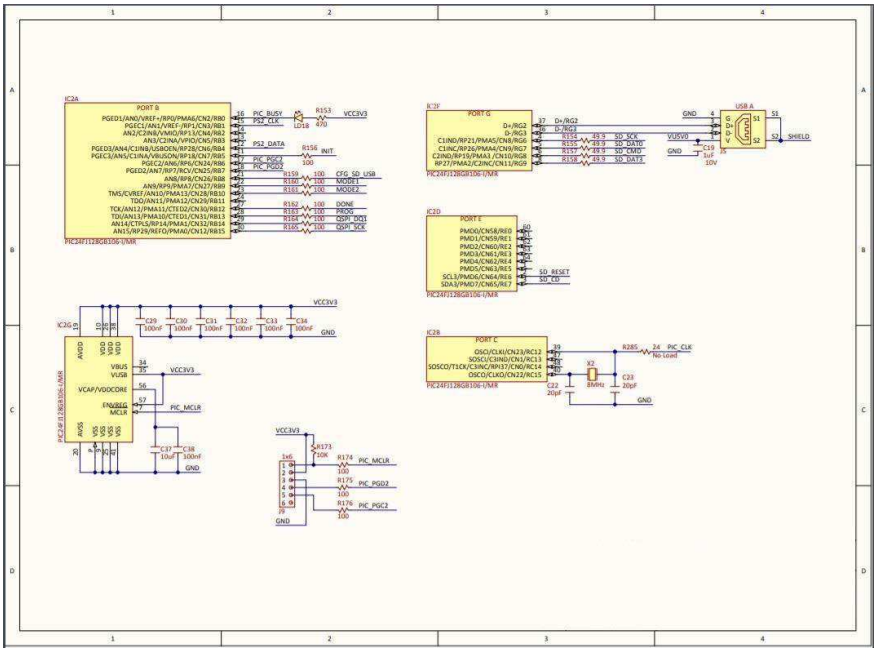
1.4 Circuito de 7-segmentos, VGA y SD



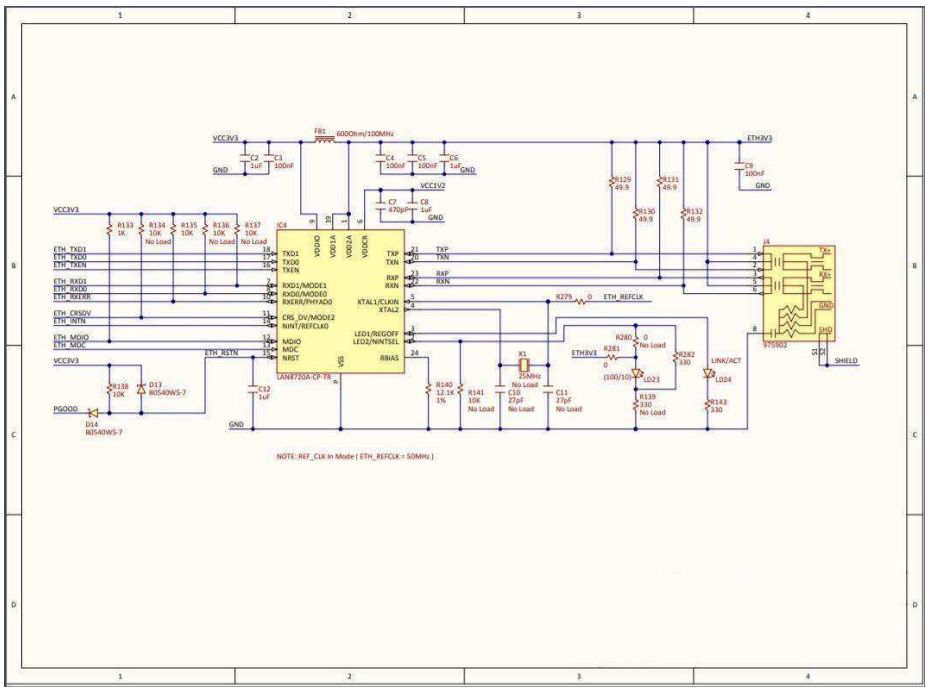
1.5 Circuito Ethernet, Temperatura, Acelerómetro y Audio



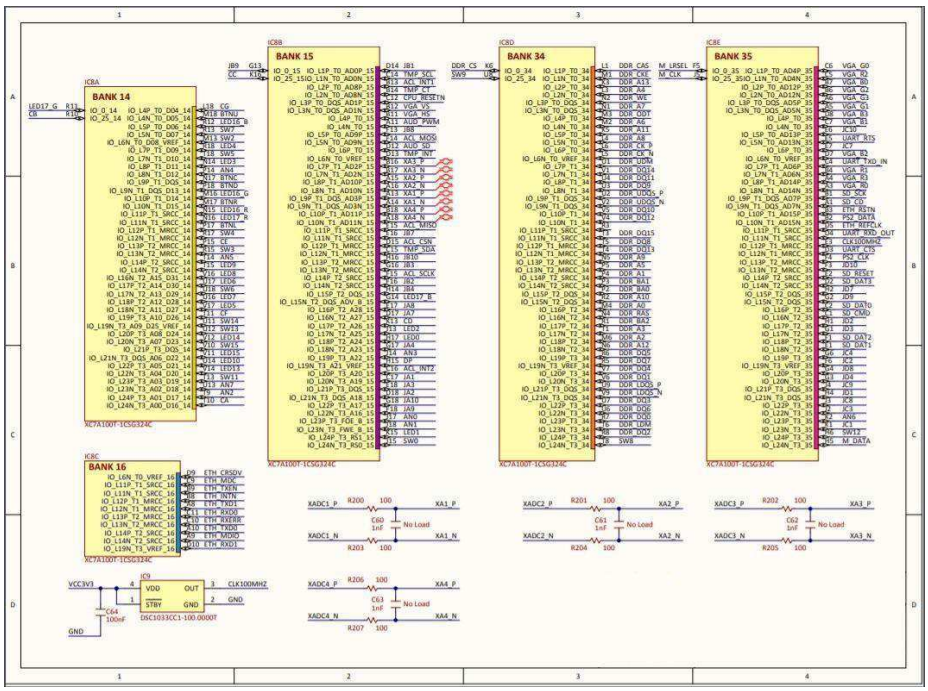
1.6 Circuito USB HID



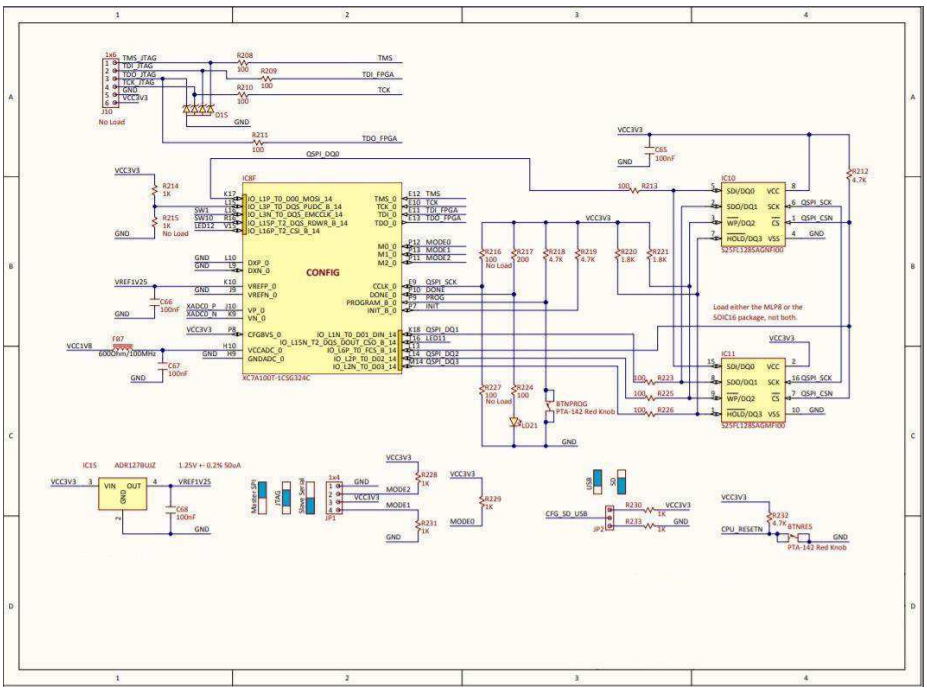
1.7 Circuito Ethernet



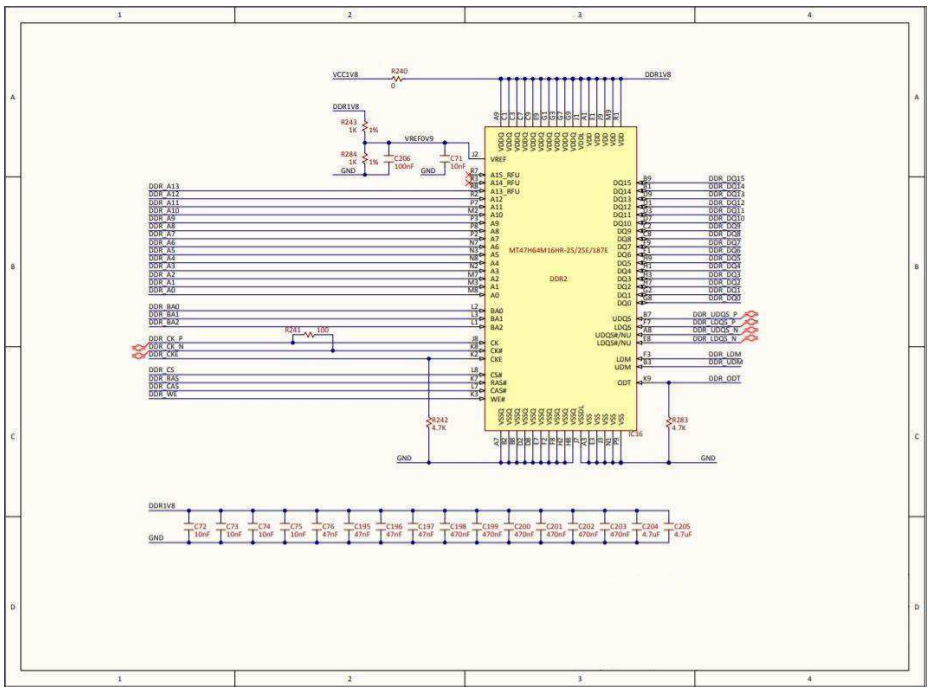
1.8 Circuiería de los Bancos de la FPGA



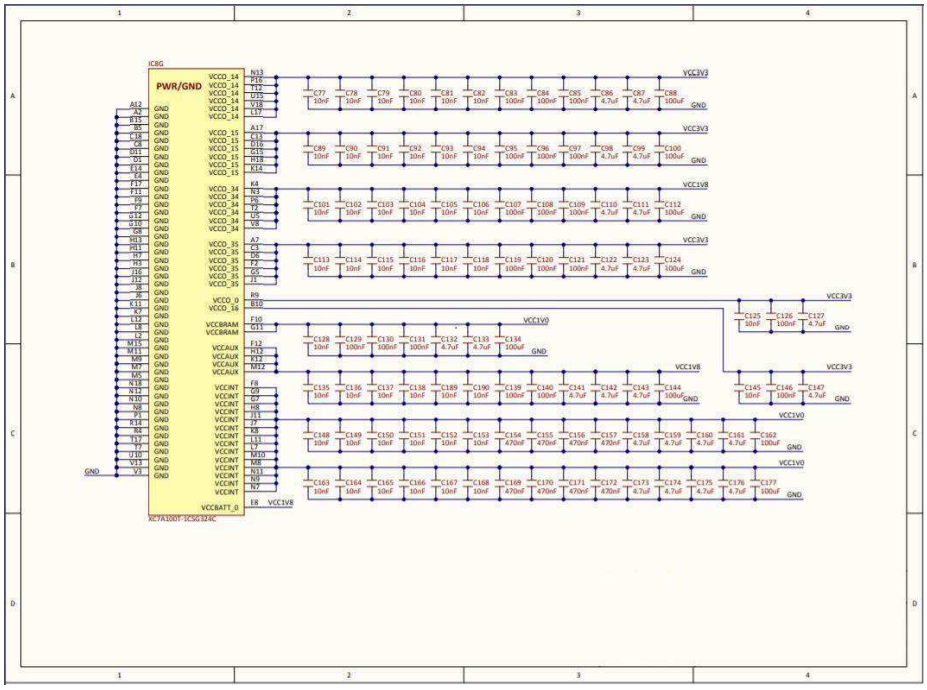
1.9 Circuito de la configuración SPI Flash



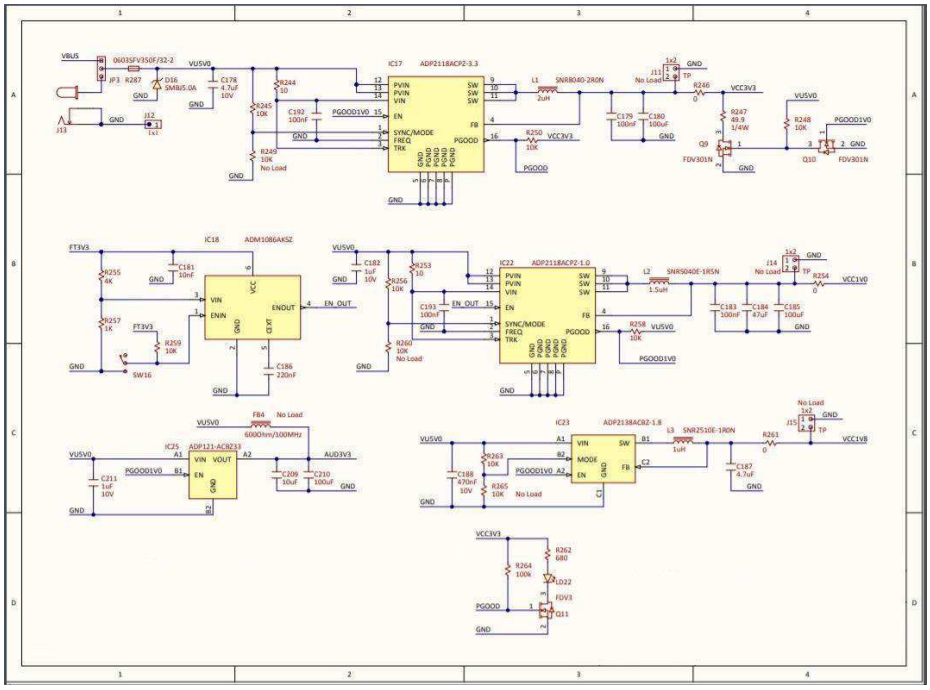
1.10 Ciruito de Memoria DDR2



1.11 Circuito de activación de la FPGA



1.12 Circuito de activación



Estos circuitos o documentos gráficos se pueden obtener desde los sitios web señalados en el apartado 7.3 de *Bibliografía* en los puntos [3] y [5], ya que provienen de la propia dirección web de Digilent que es la empresa que crea y distribuye los productos.

PLIEGO DE CONDICIONES

Índice

Índice de Ilustraciones

Índice de tablas

1. Condiciones Hardware

Para poder lograr la realización de completa del TFG, los elementos que se necesitan son:

- Tarjeta Nexys 4 DDR de Xilinx.
- Cable con configuración USB tipo A – Micro USB tipo B para conexión entre un PC y la tarjeta

Es necesario guardar los archivos y carpetas de los proyectos con nombres sencillos y que no contengan caracteres especiales, ya que podrán dar errores a la hora de sintetizar o abrir algunos archivos.

1.1 Condiciones de alimentación

A continuación se muestra una tabla referida a los distintos suministros de tensión e intensidad que debe adoptar la tarjeta Nexys 4 DDR dependiendo de los circuitos que se vayan a utilizar:

Fuentes de alimentación Nexys 4 DDR			
Suministro	**Circuitos**	**Dispositivo**	**Corriente**
3.3 V	E/S FPGA, puertos USB, relojes E/S RAM, Ethernet, ranura SD, sensores flas	IC17: ADP2118	3A / 0.1 a 1.5A
1.0 V	FPGA Core	IC22: ADP2118	3A / 0.2 a 1.3A
1.8 V	DDR2, FPGA Auxiliar y RAM	IC23: ADP2118	0.8A / 0.5 A

Tabla 1: *Suministros requeridos de la tarjeta Nexys 4 DDR*

1.2 Condiciones térmicas

Se trata de una tarjeta de programación que dependiendo del cometido para el cuál se emplea puede se necesaria una ubicación más protegida.

En cuanto a las condiciones de temperatura de almacenamiento, el rango límite

que no debe superarse es el comprendido entre -65ºC y 150ºC. La temperatura de conexión recomendable para dispositivos comerciales queda comprendida entre 0ºC y 85ºC.

En caso de que tenga que ser ubicada en el exterior, es necesario proteger la tajeta, ante condiciones climatológicas adversas, en un lugar cubierto que disponga de ventilación para evitar subidas de temperatura por encima de los límites indicados en este mismo apartado.

1.3 Conexiones

Es necesario un conector con configuración USB tipo A – Micro USB tipo B para conectar la tarjeta Nexys 4 DDR a un PC para transferir el archivo de programación y ejecutar el programa en la tarjeta.

El conector viene incluido con la propia tarjeta Nexys 4 DDR al realizar su compra.

2. Condiciones Software

El software necesario para desarrollar la parte práctica de este TFG consta de:

- Programa Vivado 2018.3 de Xilinx, para la programación en lenguaje VHDL de la estructura hardware del proyecto.
- Programa SDK 2018.3 de Xilinx, para la programación en lenguaje C/C++ de la parte software del proyecto.
- Programa emulador de terminal de consola PuTTy, para la visualización de la transmisión por puerto serie por medio de la UART.

3. Condiciones de Usuario

3.1 Manual de Instrucciones

Uno de los objetivos principales de este trabajo, consistía en la realización de un manual de instrucciones para los usuarios que se propongan realizar un proyecto de este tipo, junto con la demo desarrollada en la sección 6 de *Resultados finales*.

Se trata un manual que aplica una serie de condiciones a seguir para lograr el cometido final de desarrollar un sistema On – Chip con MCS de 32 bits sobre la FPGA Nexys 4 DDR de Xilinx. Este manual se divide en dos partes: la parte hardware, en la que se describe el funcionamiento y programación mediante el programa *Vivado 2018.3*, y la parte software, la cual indica la programación con *SDK 2018.3* de forma estructurada.

3.1.1 Flujo de desarrollo hardware

La parte hardware, se explica a continuación y distingue una serie de pasos principales que son los siguientes:

1. Crear un proyecto de diseño.
2. Agregar o crear instancias centrales de Xilinx IP
3. Agregar o crear códigos de diseño HDL.
4. Agregar un archivo de restricción.
5. Realizar síntesis, implementación y generación de bitstream.
6. Programar un dispositivo FPGA.

A continuación, se van a desarrollar de forma detallada y con la ayuda de imágenes o capturas cada uno de los pasos anteriores.

3.1.1.1 Crear un proyecto de diseño

Un nuevo proyecto en Vivado comprobado hasta la versión 2018.3, se puede crear de la siguiente manera:

1.- Seleccionar *Vivado* en el menú de inicio de Windows o haciendo clic en el icono de *Vivado*.

2.- En la ventana de *Inicio de Vivado*, hacer clic en el icono *Create New Project* (Crear nuevo proyecto*)*. Aparece la ventana Nuevo proyecto.

3.- Ingresar el nombre del proyecto deseado (por ejemplo: TFG_demo_final) y la ubicación de directorio deseada y hacer clic en *Next* (Siguiente).

4.- En el cuadro de diálogo *Project Type* (Tipo de proyecto), seleccionar *RTL Project* (proyecto RTL) y marcar la casilla *Do not specify sources at this time* (No especificar fuentes en este momento). Hacer clic en *Next*. Agregar los archivos más adelante con *Project Manager*, que es más flexible y brinda más control.

5.- En el cuadro de diálogo de selección de partes, hacer clic en la pestaña *Parts* (Partes) en el campo *Select* para especificar el dispositivo FPGA de destino. Para la placa *Nexys 4 DDR*, seleccionar lo siguiente:

- Product Category: all
- Family: Artix-7
- Sub-familiy: Artix-7
- Package: csg324
- Speed: -1
- Part: xc7a100tcsg324-1

Se muestra en inglés debido a que el idioma del programa Vivado no se encuentra en español y son los campos que un usuario se va a encontrar tal cual.

Para otras placas, esta información se puede encontrar en el manual de la placa FPGA o en la marca en la parte superior del chip FPGA. Después de la selección, hacer clic en *Next* y luego en *Finish* para completar la creación. Aparece la ventana principal de Vivado, similar a la de la Ilustración PL.1.

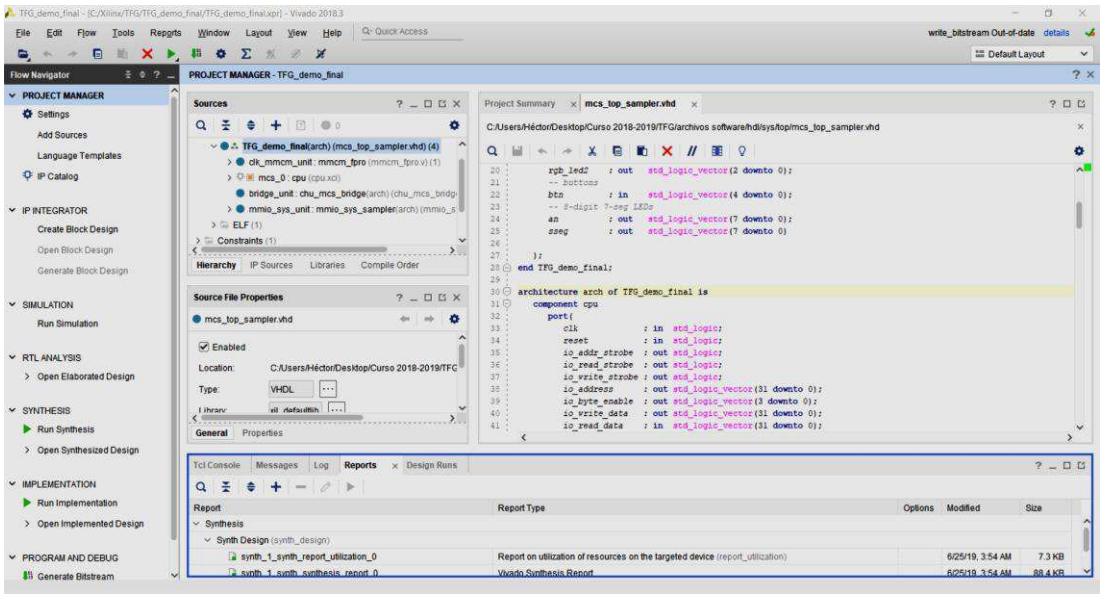


Ilustración PL.1: *Ventana típica de Vivado.*

La información del dispositivo y el idioma se puede cambiar más adelante (no es posible cambiar a español) invocando el subproceso de *Project Settings* (Configuración del proyecto) en la subventana del *Flow Navigator* (Navegador de flujo).

3.1.1.2 Agregar o crear instancias centrales de Xilinx IP

Xilinx proporciona una colección completa de núcleos IP. En el enfoque de este tutorial no se utiliza ninguna instancia de núcleo de Xilinx IP, dado que el trabajo está basado en el diseño de hardware digital, y se desarrollan la mayoría de los circuitos en HDL desde cero. El procedimiento para agregar archivos de instancias IP existentes a un proyecto es similar a agregar archivos HDL existentes, pero no entra dentro de las intenciones de este trabajpo.

3.1.1.3 Agregar o crear códigos de diseño HDL

Después de crear un proyecto, se pueden agregar al proyecto archivos de la parte hardware en lenguaje HDL existentes o crear nuestros propios archivos desde cero. El procedimiento para agregar archivos HDL existentes es el siguiente:

1.- En la subventana de *Flow Navigator*, expandir *Project Manager* y luego seleccionar *Add Sources* (Agregar Fuentes). Aparece un diálogo.

2.- Seleccionar el botón *Add or create design sources* (Agregar o crear fuentes de diseño) y hacer clic en *Next* para pasar al siguiente cuadro de diálogo.

3.- En esta ventana de diálogo, presionar el signo "+" en la mitad de la ventana de diálogo. Aparece una pequeña ventana con tres elementos: *Add files*..., *Add directories*... y *Create files*...

4.- Hacer clic en el elemento *Add Files…* (Agregar archivos...) y navegar hasta la ubicación. Seleccionar los tres archivos comparadores y agregarlos a la lista. Alternativamente, si los tres archivos están en la misma carpeta, hacer clic en *Add Directories…* para agregar el directorio.

5.- Después de incluir todos los archivos necesarios, hacer clic en el botón *Finish*. Los archivos se analizarán e importarán al proyecto y se mostrarán jerárquicamente en la subventana *Design Sources*, como se muestra en la Ilustración PL.2.

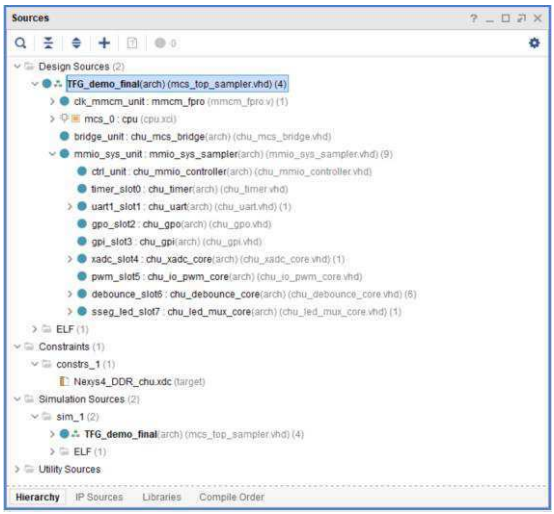


Ilustración PL.2: *Subventana de fuentes típica de Vivado.*

Alternativamente, los archivos HDL pueden construirse desde cero. El procedimiento para crear un nuevo HDL es el siguiente:

1.- Seguir los primeros tres pasos como antes.

2.- Hacer clic en el elemento *Create Files…* (Crear archivos) y aparecerá el cuadro de diálogo *Create Source File* (Crear archivo fuente).

3.- Establecer el campo *File type* (Tipo de archivo) en VHDL, ingresar el nombre del archivo y hacer clic en *OK* para cerrar el cuadro de diálogo.

4.- Hacer clic en *Finish* para cerrar el cuadro de diálogo *Add Sources* (Agregar fuentes). Aparece un cuadro de diálogo *Define Module* (Definir módulo).

5.- Este diálogo nos permite ingresar los nombres de los puertos y el nombre de la arquitectura. Estos nombres se insertan en el código HDL más adelante. Hacer clic en *OK* y el archivo se agregará a la subventana de *Sources*.

6.- Hacer clic en el archivo y aparecerá en la subventana *Workplace* (Lugar de trabajo). Ingresar el código HDL y luego guardar el archivo.

3.1.1.4 Agregar un archivo de restricción

Las restricciones son ciertas condiciones impuestas en los procesos de síntesis e implementación. Para nuestros propósitos, las principales limitaciones son las asignaciones de pines de los puertos de E/S y la velocidad de reloj del sistema. Durante el proceso de implementación, las señales de E/S de cada módulo deben asignarse a un pin físico del dispositivo FPGA. Dado que las señales de E/S de los periféricos ya están conectadas permanentemente a los pines del FPGA designado en la placa de prototipos, debemos asegurarnos de que las señales se asignen a los pines correspondientes. El otro tipo de restricción tiene que ver con el tiempo, que especifica la frecuencia de reloj del oscilador de la placa.

La información sobre restricciones se presenta en formato XDC (Xilinx Design Constraints o restricciones de diseño de Xilinx), que se basa en el formato estándar de la industria SDC (Synopsys Designs Constraints o restricciones de diseño de sinopsis).

La información de restricción se almacena en un archivo con una extensión de *.xdc* y puede ser editada por un editor de texto normal.

Para este trabajo se ha utilizado el archivo de restricción `nexys4_ddr_chu.xdc`. El archivo está diseñado para la placa DDR de Nexys 4 e incluye las restricciones para las asignaciones de pines y la velocidad de reloj del sistema. Se recomienda utilizar los mismos nombres de puerto de E/S en el módulo principal.

El procedimiento para agregar el archivo de restricción a un proyecto es similar al de agregar un archivo de diseño:

1.- En la subventana de Flow Navigator, expandir Project Manager y luego seleccionar Add Sources. Aparece un diálogo.

2.- Seleccionar el botón *Add or create constraints* y hacer clic en *Next* para pasar al siguiente cuadro de diálogo.

3.- En esta ventana de diálogo, presionar el signo "+".

4.- Hacer clic en *Add files...* y navegar hasta la ubicación. Seleccionar el archivo `nexys4_ddr_chu.xdc` y verificar el cuadro de *copy constraints into project*.

5.- Hacer clic en el botón *Finish*. El archivo se importará al proyecto y se mostrará en la carpeta *Constraints* de la subventana de *Sources*.

6.- Hacer clic en el archivo, aparecerá en la subventana *Workplace*. Comentar las restricciones asociadas a señales de E/S no utilizadas y guardar el archivo.

El último paso puede ser omitido. Sin embargo, esto no se recomienda ya que las asignaciones de pin no utilizadas en el archivo de restricciones conducirán a un mayor número de mensajes de advertencia.

3.1.1.5 Realizar síntesis, implementación y generación de bitstream

La realización de un diseño consta de tres procesos en cascada:

- Síntesis
- Implementación
- Generación Bitstream

Los procesos se pueden invocar desde la subventana de Flow Navigator secuencialmente. El procedimiento es el siguiente:

1.- Asegurarse de que el módulo deseado esté designado como el módulo de nivel superior (resaltado con negrita y un icono a la izquierda).

2.- En la subventana de *Flow Navigator*, expandir *Synthesis* (Síntesis) y luego seleccionar *Run synthesis* (Ejecutar síntesis).

3.- Si hay errores, revisar la pestaña *Message* (Mensajes) en el área de *Console* (consola), solucionar los problemas y repetir.

4.- Si no hay un error de síntesis, aparece una ventana de *Synthesis Completed* (Síntesis completada) y solicita la siguiente acción. Hacer clic en el botón *Cancel* (Cancelar) ya que invocaremos los procesos subsiguientes manualmente.

5.- Se recomienda revisar los mensajes de advertencia. Muchos mensajes se relacionan con errores de diseño, como señales sin asignar y bucles de combinación. Arreglar los problemas y repite el paso 2.

6.- Una vez completado el proceso de síntesis, se generan una serie de análisis e informes. Si se desea, expandir el subproceso de *Synthesized Design* (Diseño sintetizado) y luego seleccionar y examinar un informe.

7.- En la subventana de Flow Navigator, expandir Implementación y luego seleccionar *Run Implementation* (Ejecutar implementación).

8.- Si se desea, expandir el subproceso *Open Implemented Design* (Abrir diseño implementado) y seleccionar y examinar un informe.

9.- En la subventana del *Flow Navigator*, expandir *Program and Debug* (Programar y Depurar) y luego seleccionar *Generate Bitstream* (Generar Bitstream). Cuando se completa el proceso, aparece el cuadro de diálogo *Bitstream Generation Completed* (Generación de flujo de bits completado). Hacer clic en el botón *Cancel* para cerrar el cuadro de diálogo.

Recordar que Flow Navigator admite el esquema de "creación automática", que ejecuta automáticamente los procesos necesarios para llegar al paso deseado. Otra forma de completar estas tareas es simplemente iniciar el proceso de

generación de bitstream. Los procesos de síntesis e implementación serán invocados automáticamente.

3.1.1.6 Programar un dispositivo FPGA

El último paso es programar el dispositivo FPGA; es decir, descargar el archivo de configuración (el archivo *.bit*) al dispositivo FPGA. El procedimiento es el siguiente:

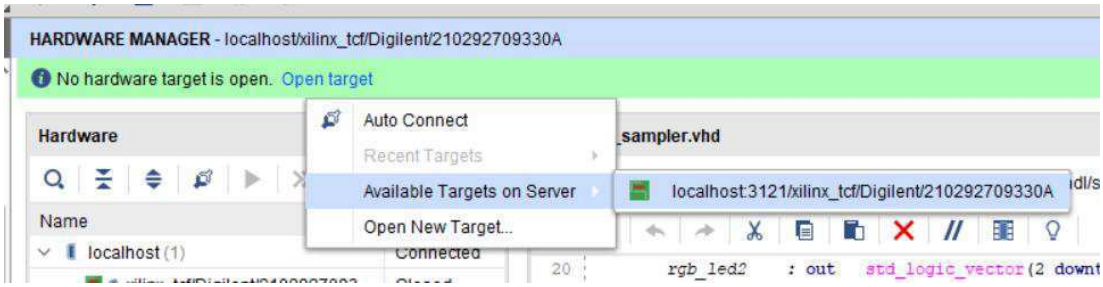


Ilustración PL.3: *Captura de pantalla de conexión con la tarjeta Nexys.*

1.- Conectar el cable USB al puerto micro USB (etiquetado *PROG UART*) en la tarjeta Nexys 4 DDR y encender la alimentación. Asegurarse de que el puente *JP1* esté en la posición *JTAG*.

2.- En la subventana de *Flow Navigator*, expandir *Program and Debug*, seleccionar *Hardware manager* (Administrador de hardware) y luego selecciónar *Open Target* (Abrir objetivo). Aparece una pequeña ventana, como se muestra en la Ilustración PL.3. Si la tarjeta Nexys 4 DDR se configuró antes, aparece como localhost: ... Seleccionarla e ir al Paso 4. A continuación, aparecerá la ventana *Open New Target* (Abrir nuevo objetivo) y *Open New Hardware Target* (Abrir nuevo objetivo de hardware).

3.- Hacer clic en el botón *Next* para pasar por una serie de pantallas. Seleccionar *Local server (*Servidor local) en la segunda pantalla y seleccionar *Digilent board* (Tablero de Digilent) en la tercera. Hacer clic en el botón *Finish* para cerrar la ventana.

4.- En la subventana de *Flow Navigator*, seleccionar *Program Device* (Dispositivo del programa). Aparece una pequeña ventana etiquetada con *xc7a100t*. Seleccionarla y aparecerá un diálogo de *Program Device*.

5.- Hacer clic en el botón *Program* para descargar el archivo *.bit*.

6.- Comprobar el correcto funcionamiento de la tarjeta manejando las entradas de swtiches, botones, etc, y observando los resultados en las salidas programadas.

Una forma alternativa de configurar el FPGA es descargar el archivo de configuración en un dispositivo flash y cargar el archivo de configuración cuando se enciende. Se puede encontrar más información sobre este método en el sitio web de Digilent.

3.1.2 Desarrollo del sistema FPro

El desarrollo del sistema FPro completo implica el desarrollo en conjunto de hardware y software. El procedimiento consta de los siguientes pasos:

1.- Crear un proyecto de diseño.

2.- Agregar o crear una instancia de MicroBlaze MCS.

3.- Agregar o crear códigos HDL con una instancia de MCS.

4.- Agregar un archivo de restricción.

5.- Realizar síntesis, implementación y generación de bitstream.

6.- Exportar la configuración del hardware.

7.- Desarrollar la parte software y generar el archivo ejecutable (archivo *.elf*).

8.- Añadir el archivo *.elf* en el módulo de memoria de FPGA y regenerar archivo *bitstream*.

9.- Configurar un programa emulador de terminal.

10.- Programar un dispositivo FPGA.

Se amplía el procedimiento de desarrollo de hardware anterior en la sección 3.1 de Anexos, e incorpora tres pasos adicionales (Pasos 6, 7 y 8) para acomodar el desarrollo de software. Los tutoriales de los tres pasos se proporcionan en las siguientes subsecciones.

Hay que tener en cuenta que Vivado Desing Suite puede servir como plataforma

para el desarrollo de SoC. El proceso de integración de IP de Flow Navigator es para este propósito. Sin embargo, la plataforma está diseñada para núcleos IP basados en MicroBlaze y AXI con todas las funciones. El soporte para MicroBlaze MCS es limitado y su desarrollo no sigue el flujo general basado en IP de Vivado. El sistema FPro en este trabajo está construido desde cero y no utiliza ninguna de las funciones integradas de integración de IP de Vivado.

3.1.2.1 Configuración del hardware de exportación

Para establecer la unión entre la parte hardware programada mediante Vivado 2018.3 y la parte software programada con SDK 2018.3 (como se va a guiar más adelante) es necesario configurar el hardware realizado en la sección 3.1 anterior. Para ello es necesario realizar los siguientes pasos:

1.- Seleccionar el menú *File -> Export -> Export Hardware*... y aparecerá una subventana.

2.- En el campo *Export to*, navegar a la carpeta de destino y luego hacer clic en el botón *OK*.

3.- El archivo de descripción de hardware (con la extensión de .hdf) se genera en la carpeta designada.

Dado que un sistema FPro está diseñado manualmente desde cero y no desde *IP Integrator*, el archivo de descripción de hardware solo contiene la información sobre la configuración de *MicroBlaze MCS*, no todo el sistema FPro. Por lo tanto, este paso no necesita repetirse si se utiliza la misma instancia de MicroBlaze MCS.

3.1.2.2 Desarrollo del software

Se utiliza *Xilinx SDK* (*Software Development Kit* o kit de desarrollo de software) versión 2018.3 como plataforma para el desarrollo de software. En la Ilustración PL.4 se muestra una ventana típica del SDK de Xilinx.

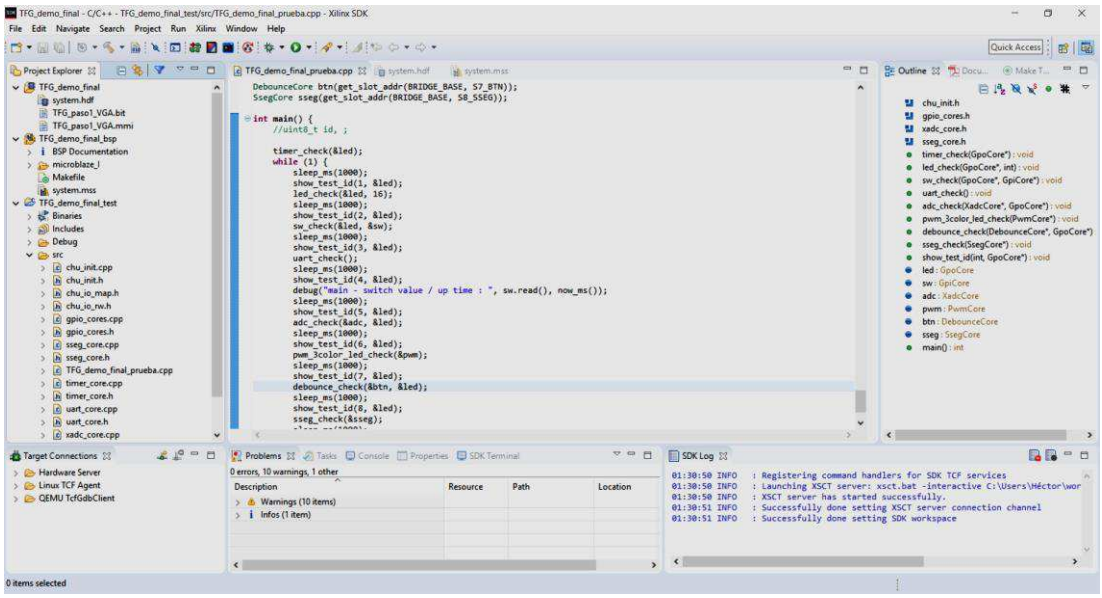


Ilustración PL.4: *Ventana típica de Xilinx SDK.*

El modelo de software básico constituye una jerarquía de tres capas:

- Especificaciones de la plataforma de hardware
- *BSP* (*Board Support Package* o paquete de soporte a bordo)
- Programa de aplicación

Un sistema integrado se basa en una aplicación específica y su configuración se adapta para admitir la aplicación. La especificación de la plataforma de hardware es la capa inferior que recoge la información de hardware relevante requerida para el desarrollo y la implementación del software. *BSP* es la capa intermedia. Es una biblioteca de software que contiene controladores y rutinas de inicio basadas en la información de un archivo de especificación de plataforma de hardware específico. El término se toma del tradicional desarrollo de aplicaciones integradas, en el que el sistema generalmente se realiza mediante una placa de circuito impreso personalizado. Un programa de aplicación es la capa superior que utiliza las rutinas de la biblioteca BSP para acceder al hardware.

Este modelo de software está automatizado para un sistema derivado del Integrador de IP de Vivado. Como el sistema FPro está diseñado desde cero, las dos primeras capas son solo para *MicroBlaze MCS* y el *BSP* solo contiene una rutina de inicio. Necesitamos incluir manualmente los controladores de E/S del

software en un programa de aplicación.

El desarrollo del software construye las tres capas en secuencia. Los pasos básicos son los siguientes:

1.- Seleccionar *Xilinx SDK* en el menú de inicio de Windows o hacer clic en el icono de *SDK*. No lo lanzar desde vivado.

2.- Crear o seleccionar un espacio de trabajo.

3.- Seleccionar *File -> New -> Others ->* y aparecerá una ventana. Expandir la carpeta Xilinx y luego seleccionar *Hardware Platform Specification* (Especificación de plataforma de hardware). Aparece el cuadro de diálogo *New Hardware Project* (Nuevo proyecto de hardware).

4.- Ingresar un nombre, por ejemplo `TFG_demo_final`, para el proyecto de hardware. En el campo *Target Hardware Specification* (Especificación de hardware de destino), navegar a la carpeta de destino especificada en la sección 3.2.1 y seleccionar el archivo *.hdf*. Hacer clic en *Finish* para generar las especificaciones de la plataforma de hardware. La nueva carpeta de especificaciones aparece en la subventana de *Project Explorer* (Explorador de proyectos).

5.- Seleccionar *File -> New -> Board Support Package* (Paquete de soporte de placa) y aparecerá un cuadro de diálogo. Ingresar un nombre, por ejemplo `TFG_demo_final_bsp`, para el proyecto BSP, seleccionar la `TFG_demo_final` creada anteriormente en *Target Hardware* y luego seleccionar la opción *standalone* (independiente) para el campo OS, como se muestra en la Ilustración PL.5. Hacer clic en Finalizar para generar BSP. La nueva carpeta BSP aparece la subventana del Explorador de proyectos.

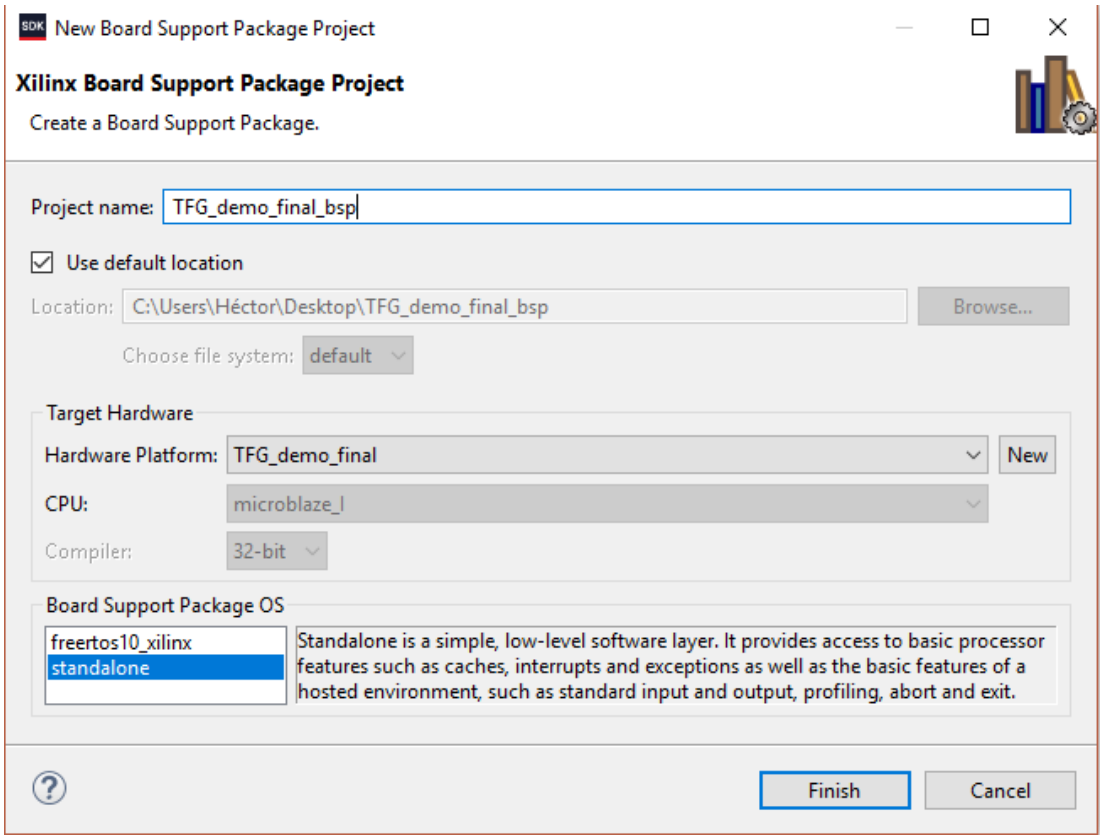


Ilustración PL.5: *Creacción del BSP en Xilinx SDK.*

6.- Seleccionar *File -> New -> Application Project* (Proyecto de aplicación) y aparecerá un cuadro de diálogo. Ingresar un nombre, por ejemplo `TFG_demo_final_test`, para el proyecto de la aplicación, seleccionar el `TFG_demo_final` y `TFG_demo_final_bsp` creados anteriormente y luego hacer clic en el botón C ++, como se muestra en la Ilustración PL.6. Hacer clic en Finalizar para crear un nuevo proyecto de aplicación. La nueva carpeta de proyectos de aplicaciones aparece en la subventana del Explorador de proyectos.

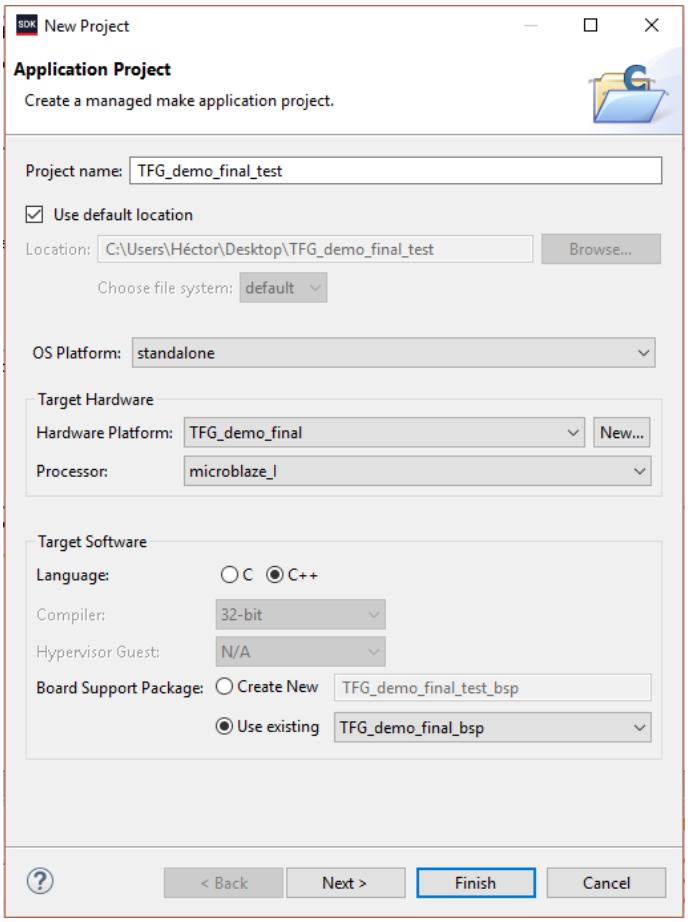


Ilustración PL.6: *Creacción de un Nuevo Proyecto en Xilinx SDK.*

7.- Importar el archivo principal del programa y los archivos del controlador descritos en el apartado 5.2 de *Análisis de soluciones* a la carpeta *src*. La subventana del *Project Explorer* (Explorador de proyectos) del proyecto completado se muestra en la Ilustración PL.7.

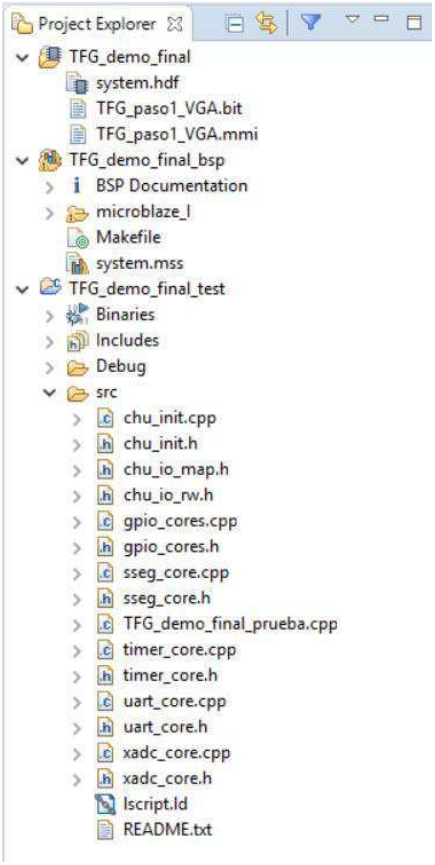


Ilustración PL.7: *Subventana del explorador del proyecto en Xilinx SDK.*

8.- De forma predeterminada, Xilinx SDK está configurado para generar un proyecto automáticamente y, por lo tanto, el archivo `TFG_demo_final_test.elf` se compila y vincula automáticamente después de que los archivos se arrastran al proyecto. Hay que tener en cuenta que el tamaño del archivo se muestra en la pestaña de la consola, que se muestra en la Ilustración PL.8.

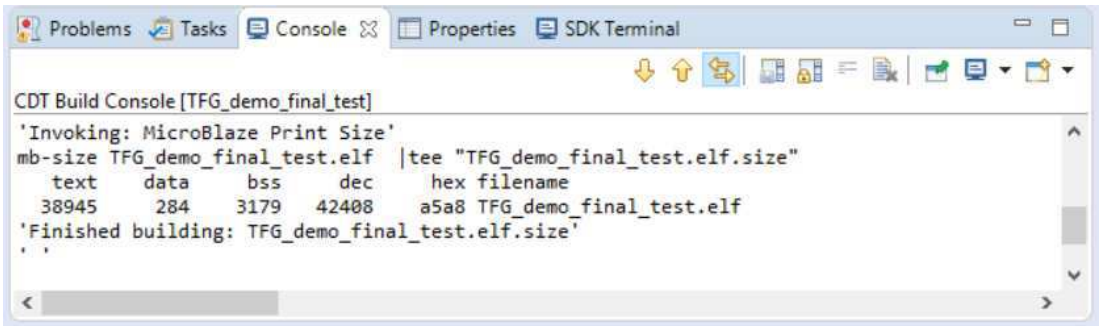


Ilustración PL.8: *Información del tamaño del archivo en Xilinx SDK.*

3.1.2.3 Integrar el archivo .elf y regenerar el archivo *.bit*

Una vez que se genera el archivo *.elf*, puede incorporarse en las definiciones del módulo. Cuando se genera bitstream, estos valores se incrustan en el archivo *.bit*. Los pasos son los siguientes:

1.- En Vivado Design Suite, seleccionar el menú *Tools -> Associate ELF Files…* (Asociar archivos *ELF*) y aparecerá un cuadro de diálogo. Navegar por la carpeta y seleccionar `TFG_demo_final_test.elf` y luego hacer clic en *OK*. El archivo se mostrará en la subcarpeta *ELF* de la carpeta *Design Sources* en la subventana de *Sources*.

2.- Seguir el procedimiento en apartado 3.1.5 de la sección de *Anexos* para regenerar el archivo *.bit*.

3.1.3 Configurar un programa para un terminal de consola

Para mostrar el flujo de caracteres de salida de UART, se necesita un programa de terminal de consola. Para este trabajo se ha utilizado el programa, PuTTY. Es un cliente de telnet y se puede descargar de forma gratuita. El procedimiento para configurar PuTTY es el siguiente:

1.- Conectar la tarjeta Nexys 4 DDR al puerto USB de la PC y encender la tarjeta. El puerto del ordenador debe reconocer el dispositivo *FT232* de la placa y tratar la conexión como un puerto serie.

2.- En Windows, abrir el Panel de control, seleccionar Administrador de dispositivos y expandir Puertos (COM y LPT). La placa debe aparecer como uno de los puertos serie, etiquetado como Puerto serie USB (COM*n*), donde *n* es el número del puerto serie designado (es decir, puerto COM). Registrar este número.

3.- Ejecutar el programa PuTTY. En la ventana de su aplicación, seleccionar el botón *Serial* para el puerto serie e ingresar el correspondiente COM*n*. Asegurarse de que la velocidad de 9600 baudios esté establecida en el campo *Speed*. La pantalla de configuración completada se muestra en la Ilustración PL.9.

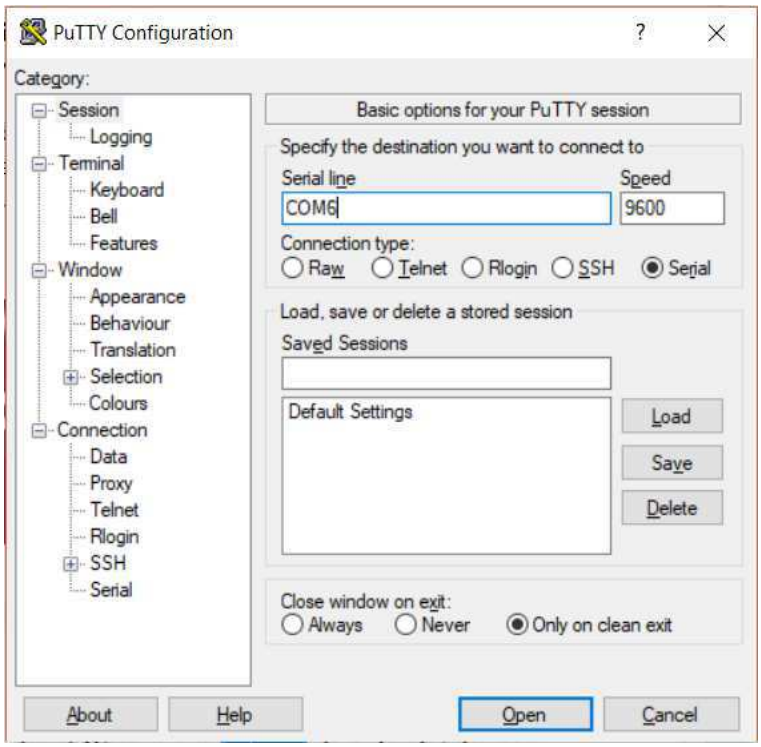


Ilustración PL.9: *Pantalla de la consola PuTTY.*

4.- Hacer clic en el botón *Open* y aparecerá la ventana del terminal donde se visualizarán los datos mostrados según la programación realizada en el proyecto.

Se concluye de esta forma, este manual de instrucciones desarrollado para poder realizar proyectos sobre aplicaciones programables SoC de 32 bits sobre FPGA, como se ha mostrado con el ejemplo de la tarjeta Nexys 4 DDR de Xilinx.

Las ilustraciones y capturas de pantalla mostradas en este manual son las correspondientes al proyecto de la demo de este trabajo fin de grado que ha sido explicado en la sección 6 de Resultados finales.

Se asegura el correcto funcionamiento de dicho proyecto tipo demo, pudiéndose comprobar de forma visual el día establecido para la defensa del mismo.

Logroño, 5 de julio de 2019

El autor:

Héctor Mangado Sáenz

PRESUPUESTO

Índice

1. Presupuesto

1.1 Unidades de proyecto

Sección	Nombre	Referencia
Recursos Hardware	Nexys 4 DDR	HW001
	Torre ordenador	HW002
	Pantalla Monitor	HW003
	Teclado	HW004
	Ratón	HW005
Recursos Software	Vivado 2018.3	SW001
	SDK 2018.3	SW002
	PuTTy	SW003
	Microsoft Office	SW004
Horas invertidas	Estudio y análisis	HI001
	Búsqueda de información	HI001
	Búsqueda de conceptos desconocidos	HI002
	Desarrollo del documento	HI003
	Descarga de Hw y Sw	HI004
	Programación Hardware	HI005
	Programación Software	HI006
	Realización de pruebas	HI007

1.2 Precio Unitario

Sección	Nombre	Referencia	Precio Unitario
Recursos Hardware	Nexys 4 DDR	HW001	600 €
	Torre ordenador	HW002	
	Pantalla Monitor	HW003	125 €
	Teclado	HW004	20 €
	Ratón	HW005	15 €
Recursos Software	Vivado 2018.3	SW001	0 €
	SDK 2018.3	SW002	0 €
	PuTTy	SW003	0 €
	Microsoft Office	SW004	0 €
Horas invertidas	Estudio y análisis	HI001	20 €/h
	Búsqueda de información	HI001	
	Búsqueda de conceptos desconocidos	HI002	
	Desarrollo del documento	HI003	20 €/h
	Descarga de Hw y Sw	HI004	
	Programación Hardware	HI005	50 €/h
	Programación Software	HI006	
	Realización de pruebas	HI007	

1.3 Mediciones

Sección	Nombre	Referencia	Unidades
Recursos Hardware	Nexys 4 DDR	HW001	1
	Torre ordenador	HW002	1
	Pantalla Monitor	HW003	1
	Teclado	HW004	1
	Ratón	HW005	1
Recursos Software	Vivado 2018.3	SW001	1
	SDK 2018.3	SW002	1
	PuTTy	SW003	1
	Microsoft Office	SW004	1
Horas invertidas	Estudio y análisis	HI001	165 h
	Búsqueda de información	HI001	
	Búsqueda de conceptos desconocidos	HI002	
	Desarrollo del documento	HI003	110 h
	Descarga de Hw y Sw	HI004	
	Programación Hardware	HI005	85 h
	Programación Software	HI006	
	Realización de pruebas	HI007	

1.4 Presupuesto Parcial

Sección	Nombre	Precio Unitario	Unidades	Importe (€)
Recursos Hardware	Nexys 4 DDR + cable USB	600 €	1	600 €
	Torre ordenador	350 €	1	350 €
	Pantalla Monitor	125 €	1	125 €
	Teclado	20 €	1	20 €
	Ratón	15 €	1	15 €
Total				**1.100 €**

Asciende la citada sección de Recursos Hardware a la cantidad de MIL CIEN EUROS CON CERO CÉNTIMOS

Sección	Nombre	Precio Unitario	Unidades	Importe (€)
Recursos Software	Vivado 2018.3	0 €	1	0 €
	SDK 2018.3	0 €	1	0 €
	PuTTy	0 €	1	0 €
	Microsoft Office	0 €	1	0 €
Total				**0 €**

Asciende la citada sección de Recursos Software a la cantidad de CERO EUROS CON CERO CÉNTIMOS

Sección	Nombre	Precio Unitario	Unidades	Importe (€)
Horas Invertidas	Estudio y análisis	20 €/h	165 h	3.300 €
	Búsqueda de información			
	Búsqueda de conceptos desconocidos			
	Desarrollo del documento	20 €/h	110 h	2.200 €
	Descarga de Hw y Sw			
	Programación Hardware	50 €/h	85 h	4.250 €
	Programación Software			
	Realización de pruebas			
Total				**9.750 €**

Asciende la citada sección de Horas Invertidas a la cantidad de NUEVE MIL SETECIENTOS CINCUENTA EUROS CON CERO CÉNTIMOS

1.5 Presupuesto Total

Sección	Importe (€)	Subtotal (€)
Recursos Hardware	1.100 €	1.100 €
Recursos Software	0 €	0 €
Horas Invertidas	9.750 €	10.860 €
Total	10.860 €	10.860 €
Total con IVA	2.280,60 €	13.140,60 €
Total con beneficio Industrial (6%)	651,60 €	**13.792,20 €**

Asciende el trabajo desarrollado a la cantidad total de TRECE MIL SETECIENTOS NOVENTA Y DOS EUROS CON VEINTE CÉNTIMOS

Logroño, 5 de julio de 2019

El autor:

Héctor Mangado Sáenz

Notas al presupuesto:

- Los precios utilizados para este proyecto son los vigentes a fecha de junio de 2019. En el supuesto de que transcurran más de 12 meses hasta su ejecución los preciós deberían ser actualizados.
- Las versiones de software utilizadas corresponden a las disponibles en junio de 2019. Se ha de tener en cuenta posibles actualizaciones y nuevas versiones.

Logroño, 5 de julio de 2019

El autor:

Héctor Mangado Sáenz